

# JAVA™

---

## 技术手册

第五版

*David Flanagan* 著  
*O'Reilly Taiwan* 公司 编译

O'REILLY®

*Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo*

O'Reilly Media, Inc. 授权东南大学出版社出版

东南大学出版社

## 图书在版编目 (CIP) 数据

Java 技术手册: 第 5 版 / (美) 弗拉纳根 (Flanagan, D.)  
著; O'Reilly Taiwan 公司编译. —南京: 东南大学出版社,  
2006.10

书名原文: Java in a Nutshell, fifth edition

ISBN 7-5641-0560-7

I.J... II. ①弗... ②O... III. JAVA 语言—程序设计—技  
术手册 IV. TP312-62

中国版本图书馆 CIP 数据核字 (2006) 第 109064 号

江苏省版权局著作权合同登记

图字: 10-2006-135 号

©2005 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Southeast University Press, 2005. Authorized translation of the English edition, 2005 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2005。

简体中文版由东南大学出版社出版 2005。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有。未得书面许可, 本书的任何部分和全部不得以任何形式重制。

书 名 / Java™ 技术手册 (第五版)

书 号 / ISBN 7-5641-0560-7

责任编辑 / 张烨

封面设计 / Edie Freedman, 张健

出版发行 / 东南大学出版社 (press.seu.edu.cn)

地 址 / 南京四牌楼 2 号 (邮编 210096)

印 刷 / 扬中市印刷有限公司

开 本 / 787 毫米 × 980 毫米 16 开本 26.25 印张 441 千字

版 次 / 2006 年 10 月第 1 版 2006 年 10 月第 1 次印刷

印 数 / 0001-4000 册

定 价 / 48.00 元 (册)



## O'Reilly Media, Inc. 介绍

为了满足读者对网络 and 软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权东南大学出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的 50 本书之一）到 GNN（最早的 Internet 门户和商业网站），再到 WebSite（第一个桌面 PC 的 Web 服务器软件），O'Reilly Media, Inc. 一直处于 Internet 发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。



## 作者简介

---

**David Flanagan** 是一位计算机程序员，他花了大部分的时间来编写关于 Java 和 JavaScript 的书。他在 O'Reilly 的其他书籍包括了《Java Examples in a Nutshell》、《Java Foundation Classes in a Nutshell》以及《Javascript: The Definitive Guide》。David 获得麻省理工学院计算机科学与工程学士学位。他与家人住在美国西北部，在西雅图与英属哥伦比亚省的温哥华之间。

## 封面介绍

---

本书第五版的封面是爪哇虎，它是爪哇岛特有的亚种。虽然这种虎由于其原生的孤立性而曾经提供无与伦比的研究机会，但这些机会因为人类侵入爪哇虎的栖息地而永久丧失。对这种虎的最糟糕的状况是爪哇已被开发成世界上人口最稠密的岛。在认识到此亚种的危险情况时，即使将它们关在笼子中来保护也已经太晚了。爪哇虎最后一次被看到是在 1972 年，现在它被认定已经绝种了。



# 目录

前言 .....	1
第一章 导论 .....	9
什么是 Java? .....	9
使用 Java 的好处 .....	13
程序员效率和节省时间 .....	15
第二章 Java 基本语法 .....	25
Java 概述 .....	26
词汇 (Lexical) 结构 .....	26
基本数据类型 .....	30
表达式与运算符 .....	37
语句 .....	52
Method .....	76
类与对象 .....	84
数组 .....	87
引用类型 .....	95
包与 Java 命名空间 .....	103

i

Java 文件结构 .....	107
定义并运行 Java 程序 .....	108
Java 与 C 的不同点 .....	109
<b>第三章 Java 的面向对象程序设计 .....</b>	<b>112</b>
类的定义语法 .....	113
字段与 method .....	114
对象的创建与初始化 .....	121
对象的撤消与终止 .....	126
子类与继承 .....	129
数据隐藏与封装 .....	139
抽象类与方法 .....	145
java.lang.Object 的重要 method .....	147
接口 .....	151
嵌套类型 .....	156
修饰符一览 .....	173
没有包括在 Java 中的 C++ 特性 .....	174
<b>第四章 Java 5.0 新增功能 .....</b>	<b>176</b>
泛型 (Generic Type) .....	177
枚举类型 .....	196
注释 .....	210
<b>第五章 Java 平台 .....</b>	<b>222</b>
Java 平台概述 .....	222
文本 .....	224
数值与数学运算 .....	237
日期与时间 .....	241
数组 .....	244
集合 .....	245



线程与并行 .....	259
文件与目录 .....	274
使用 java.io 输入 / 输出 .....	275
使用 java.net 进行网络连接 .....	280
使用 java.nio 进行 I/O 与网络连接 .....	285
XML .....	298
类型、反射与动态加载 .....	306
对象持久性保存 .....	309
安全性 .....	310
密码术 .....	313
各式各样的平台特色 .....	315
 <b>第六章 Java 的安全性 .....</b>	<b>321</b>
安全风险 .....	322
JAVA VM 安全性与类文件验证 .....	322
验证与加密 .....	323
访问控制 .....	323
针对所有人的安全性 .....	326
Permission 类 .....	328
 <b>第七章 程序设计与文档规范 .....</b>	<b>330</b>
命名与大小写惯例 .....	330
可移植性惯例和纯 Java 规则 .....	332
Java 说明文件的注释 .....	334
JavaBeans 惯例 .....	342
 <b>第八章 Java 开发工具 .....</b>	<b>349</b>
apt .....	349
extcheck .....	351
jarsigner .....	351

jar .....	353
java .....	356
javac .....	365
javadoc .....	369
javah .....	377
javap .....	379
javaws .....	380
jconsole .....	382
jdb .....	383
jinfo .....	388
jmap .....	389
jps .....	390
jsadebugd .....	391
jstack .....	392
jstat .....	392
jstatd .....	394
keytool .....	395
native2ascii .....	399
pack200 .....	400
policytool .....	403
serialver .....	404
unpack200 .....	405







# 前言

本书会对 Java 程序设计语言和 Java 平台的核心 API 作快速、精炼的介绍。包含了 Java 1.0、1.1、1.2、1.3、1.4 和 5.0 版本。

## 第五版中的修改

本书的第五版涵盖了 Java 5.0。就如其增加的版本编号所显示的，新版的 Java 有许多新特性。最重要的三个新语言特性是 generic、enumerated 类型以及注释 (annotation)，这些会有独立的章节来说明。只想了解这些新特性的 Java 程序员可以直接跳到第四章。

Java 5.0 的其他新语言特性是：

- 可用 `for/in` 语句轻松地处理数组和集合（此语句有时被称为“foreach”）。
- 在原值与其相对应的 wrapper 对象（例如 `int` 值和 `Integer` 对象）之间自动地来回转换的 `autoboxing` 和 `autounboxnig` 转换。
- `varargs method` 用于定义和调用接收任意数量自变量的 `method`。
- `covariant return` 能让子类改写父类 `method`，并限制 `method` 的返回类型。
- `import static` 声明将类型的 `static` 成员导入命名空间。

虽然这些特性都是 Java 5.0 中新出现的，但还没有值得单独成为一章。这些特性会集中在第二章介绍。

除了这些新语言特性之外，Java 5.0 也对 Java 平台进行了更改。重要的强化包括了以下项目：

- `java.util` 集合类已被转换为泛型，提供了对 `typesafe` 集合的支持。这个内容位于第四章。
- `java.util` 包包括了新出现的 `Formatter` 类。此类以 `printf()` 和 `format()` `method` 使 C 风格的格式化输出成为可能。相关范例位于第五章。
- 新的 `java.util.concurrent` 包包含了针对 `threadsafe` 并行程序设计的重要公用程序。第五章提供了一些范例。
- `java.lang` 有三个新的包：
  - `java.lang.annotation`
  - `java.lang.instrument`
  - `java.lang.management`
- 这些包支持在 Java 5.0 运行中的 Java 编译器的注释 (`annotation`)、安装 (`instrumentation`)、管理 (`management`) 以及监控。虽然它们在 `java.lang` 层中的位置标示出这些包非常重要，但它们不常被使用。`annotataion` 的范例位于第四章中，而 `instrumentation` 和 `management` 的范例可在第五章中找到。
- `javax.xml` 层增加了新的包。`javax.xml.validation` 支持具有模式的文件验证，`javax.xml.xpath` 支持 Xpath 查询语言，而 `javax.xml.namespace` 提供了针对 XML 命名空间的简单支持。

由于为新材料找出空间的尝试大都徒劳无功，所以我做了一些缩减。JavaBeans 标准并未包含于核心 Java API，现在似乎只与 Swing 及相关的图形 API 有关。就其本身而言，它们与本书不再有关。`java.security.acl` 包从 Java 1.2 就已被忽视，我这次已把它删去。而 `org.ietf.jgss` 包只有极少的读者有兴趣。

JavaBeans 的相关章节已从本书中删去。但是，那一章中有关 JavaBeans 命名惯例的内容仍然有用，已被移到第七章。

## 本书内容

本书介绍 Java 语言、Java 平台以及 Sun 的 Java Development Kit (JDK) 提供的 Java 开发工具。前五章是必要的，后三章涵盖了一些（但不是全部）Java 程序员感兴趣的课题。

### 第一章 导论

本章是对 Java 语言和 Java 平台的一览，说明了 Java 的重要特性和性能。它以一个 Java 范例程序作结束，并带领 Java 程序设计新手逐行进行分析。

## 第二章 Java 基本语法

本章会说明Java程序设计语言的细节，其中包括一部分Java 5.0中的语言变更。这是很长而详细的一章，并没有假设你有实际的程序设计经验。有经验的Java程序员可以把它用来当作语言参考。对于有C和C++语言经验的程序员在阅读此章时，应该能很快地学到Java语法；对于只具有些许经验的程序设计新手，应该能通过仔细地研读本章来学习Java程序设计。

## 第三章 Java 中的面向对象编程

本章描述了第二章中所说明的基本Java语法如何用来编写面向对象的Java程序。本章假设你没有程序设计的经验。程序设计新手可以把本章当作教材，有经验的Java程序员可以用来当作参考。

## 第四章 Java 5.0 语言特性

本章说明Java 5.0中的三个最大新特性：generic、enumerated类型以及annotation。如果你读过本书先前的版本，可能会想直接阅读本章。

## 第五章 Java 平台

本章是对本书涵盖的基本Java API的一览。它包含了许多简短的范例，示范如何以构成Java平台的类和接口来执行常见的工作。刚接触Java的程序员（尤其是那些通过范例就能学好的程序员）应该会发现这是有价值的一章。

## 第六章 Java 安全性

本章说明Java安全性结构，它能让未受信任的程序代码在安全的环境中运行，以避免其对宿主系统造成任何恶意的损害。所有的Java程序员都至少要对Java安全性机制有基本的熟悉，这是很重要的。

## 第七章 程序设计与文档规范

本章说明了重要且被广泛采用的Java程序设计惯例，其中包括JavaBeans命名惯例。它也说明如何通过包括特殊格式的说明文件注释，来让你的Java程序代码更具有自我说明性。

## 第八章 Java 开发工具

Sun的JDK包括了一些有用的Java开发工具，特别是Java解释器和Java编译器。本章会说明那些工具。

这八章会教你Java语言，并让你能运行Java API（译注1）。

---

译注1：中文译本已将“API Quick Reference”删除，读者可在线(<http://java.sun.com/j2se/1.5.0/docs/api/>)找到更容易检索与相互参照的API说明文件。注意，随版本的不同，网址有可能变动。



## 相关书籍

O'Reilly 出版了完整系列的 Java 程序设计书籍，其中包括了好几本与本书配套的书籍。这些配套的书籍是：

*Java Examples in a Nutshell* (中译本：《Java 实例技术手册 (第三版)》)

本书包含数以百计的完整、可运行的范例，这些范例说明了许多常见的 Java 程序设计工作，使用了 core、enterprise 和 desktop API。Java Examples in a Nutshell 的内容就像本书的第四章，但在广度和深度上作了大幅的扩充，而且所有的程序代码片段都能充分发展为可运行的范例。对于要通过体验已存在的程序代码来学习的读者来说，这是特别有价值的书。

*Java Enterprise in a Nutshell* (中译本：《Java Enterprise 技术手册》)

本书是针对 Java “Enterprise” API (例如 JDBC、RMI、JNDI 以及 CORBA) 的简洁教材。它也涵盖了例如 Hibernate、Struts、Ant、JUnit 以及 Xdoclet 等企业工具。

*J2ME in a Nutshell* (中译本：《J2ME 技术手册》)

本书是针对 Java 2 Micro Edition (J2ME) 的图形、网络以及数据库 API 的教材和快速参考。

你可以从 O'Reilly 在 <http://java.oreilly.com/> 的网站找到完整的 Java 书籍列表。像是专注于核心 Java API 的书籍，包括了：

*Learning Java* (中译本：《Java 语言学习手册 (第二版)》)，由 Pat Niemeyer 和 Jonathan Knudsen 所著

本书对 Java 作了广泛的介绍，重点是在客户端的 Java 程序设计。

*Java Swing* (中译本：《Java Swing》)，由 Marc Loy、Robert Eckstein、Dave Wood、James Elliott 和 Brian Cole 所著

本书提供了极佳的 Swing API 说明，是 GUI 开发者必读的书。

*Java Thread* (中译本：《Java 线程 (第三版)》)，由 Scott Oaks 和 Henry Wong 所著

Java 使多线程程序设计变得容易，但要把事情做好仍有点困难。本书介绍了所有你必须知道的事。

*Java I/O*，由 Elliotte Rusty Harold 所著

Java 基于流的输入/输出的结构很好。本书以它应有的详细程度加以说明。

*Java Network Programming* (中译本：《Java 网络编程 (第三版)》)，由 Elliotte Rusty Harold 所著

本书详细说明了 Java 的网络 API。

*Java Security* (中译本:《Java 安全 (第二版)》), 由 Scott Oaks 所著

本书详细说明了 Java 访问控制机制, 也说明了数字签名和消息摘要的验证机制。

*Java Cryptography*, 由 Jonathan Knudsen 所著

本书提供了关于 Java Cryptography Extension (javax.crypto.\* 包) 和 Java 中的密码学的完整说明。

## 排版约定

下面是本书使用的排版约定:

斜体字 (*Italic*)

用于强调并表示第一次使用的术语。斜体也用于表示命令、电子邮件地址、网站、FTP 站点以及文件和目录名称。

黑体 (**Bold**)

有时用于表示计算机键盘的特殊键或表示用户界面的一部分, 如 Back 按钮或 Options 菜单。

等宽字 (Constant Width)

用于表示所有的 Java 程序代码以及任何在你编程时需要逐字输入的内容, 包括关键字、数据类型、常量、method 名称、变量、类名与接口名称。

等宽斜体字 (*Constant Width Italic*)

用于表示函数参数的名称, 并且一般作为指出在程序中需要以实际值替换的项目。有时也用于表示在语句中的概念上的部分或代码行。

## 建议与评论

本书的内容都经过测试, 尽管我们做了最大的努力, 但错误和疏忽仍然是在所难免的。如果你发现有什么错误或者是对将来的版本有什么建议, 请通过下面的地址告诉我们:

美国:

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室  
奥莱理软件（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在这里找到关于本书的相关信息，包括范例程序的下载、勘误表与相关资源的链接：

<http://www.oreilly.com/catalog/javanut5> (原文书网页)  
<http://www.oreilly.com.cn/book.php?bn=7-5641-0560-7> (本书中文网页)

要想了解 O'Reilly 图书、会议、资源中心以及 O'Reilly Network 的更多信息，请访问 O'Reilly 网站：

<http://www.oreilly.com>  
<http://www.oreilly.com.cn>

## 致谢

有许多人对本书的创作提供了帮助，我十分感谢他们。许许多多在前四版中写下意见、建议、错误报告以及称赞的读者们也让我非常感激，他们的许多小贡献散布于整本书中。此外，我要对于许多提供良好建议、但未被放入此版本的读者说声抱歉。

Deb Cameron 是第五版的编辑。Deb 不但编辑了在此版中的新内容，也仔细地阅读了旧内容以做必要的更新。当我在本书的努力偏移 to 预期外的方向时，Deb 很有耐心并继续指导我回到正轨来。第四版是由 Bob Eckstein 编辑，他是个细心的编辑，非常具有幽默感。Paula Ferguson 是我的朋友兼同事，他是本书前三版的编辑。他细心的阅读和实用的建议使得本书更强大、更清楚并且更有用。

照例，本书的这个版本有个一流的技术审阅团队。Sun 的 Gilad Bracha 审阅了泛型的内容。Josh Bloch 审阅了 enumerated 类型和 annotation 方面的内容，他以前是 Sun 的员工，现在在 Google 工作。Josh 也是本书第三版和第四版的审阅者，他的帮助是提供无价的资源。Neal Gafter 回答了许多关于 annotation 和 generic 的问题，他和 Josh 一样，离开 Sun 到了 Google。SAS 的 David Biesack、韩国公司 Tmax Soft 的 Changshin Lee 以及 Tim Peierls 是我在 JSR-201 专家团队的同事，那个团队负责一些 Java 5.0 中的语言修改。他们审阅了有关 generic 和 enumerated 类型的内容。Joseph Bowbeer、Brian Goetz 以



及 Bill Pugh 是 JSR-166 或 JSR-133 专家团队的成员，并帮助我了解在 `java.util.concurrent` 包下的线程和并行问题。Sun 的 Iris Garcia 回答了关于新出现的 `java.util.Formatter` 类的问题，那个类是由他编写的。我衷心感谢这些工程师。当然，如果本书中还有任何错误，那都是我的疏忽。

第四版也是由一些来自 Sun 和其他地方的工程师审阅：Josh Bloch 审阅了 `assertion` 和 `Preferences` API 方面的内容；Bob Eckstein 审阅了 XML 方面的内容；Graham Hamilton 审阅了 `Logging` API 方面的内容；Ron Hitchens 审阅了 `New I/O` 方面的内容；Jonathan Knudsen（他是个 O'Reilly 作者）审阅了 `JSSE` 和 `Certification Path` 方面的内容；Charlie Lai 审阅了 `JAAS` 方面的内容；Ram Marti 审阅了 `JGSS` 方面的内容；Philip Milne 原先是 Sun 的员工，现在在 `Dresdner Kleinwort Wasserstein`，他审阅了 `JavaBeans persistence` 机制的内容；Mark Reinhold 审阅了 `java.nio` 方面的内容；我要特别感谢 Mark，他审阅了本书的第二、第三和第四版；Andreas Sterbenz 和 Brad Wetmore 审阅了 `JSSE` 方面的内容。

第三版也从非常熟悉 Java 平台的审阅者处得到了极大的益处。Joshua Bloch 是 `Java collection framework` 的原始作者之一，他审阅了我对 `collection` 类和接口的描述。Josh 在 `Java 1.3` 的 `Timer` 和 `TimerTask` 类上也非常帮忙。Mark Reinhold 是 `java.lang.ref` 包的创建者，他对我说明了这个包，并审阅我在这部分所写的内容。Scott Oaks 审阅了关于 `Java` 安全性和密码学的类和接口的描述。`javax.crypto` 包及其分包的说明也是由 Jon Eaves 审阅。最后，第一章依据还不熟悉 `Java` 平台的审阅者的意见做了改善：Christina Byrn 以程序设计新手的角度做了审阅，而 `Virginia Power` 的 Judita Byrne 以专业 `COBOL` 程序员的身份提供了意见。

对于第二版，John Zukowski 审阅了我写的 `Java 1.1 AWT` 快速参考内容，而 George Reese 审阅了大部分余下的新资料。第二版也有幸得到来自 Sun 的技术检阅者所组成的“梦幻团队”的帮助。John Rose 是 `Java` 内部类规范内容的作者，审阅了关于内部类的章节。Mark Reinhold 是 `java.io` 中新的字符流类内容的作者，审阅了关于这些类的说明。Nakul Saraiya 是 `Java Reflection API` 的设计者，审阅了关于 `java.lang.reflect` 包的说明。

Mike Loukides 对本书的第一版提供了高水平的说明和指导。Eric Raymond 和 Troy Downing 审阅了第一版——他们帮助找出我的错误和疏忽，并提供了良好的建议，让本书对 `Java` 程序员更有用。

在将我提交的电子文件制作成的书籍中，O'Reilly 的制作团队也创造了极佳成果。我非常感谢他们。

一如往常，我要把我的感激和爱献给 Christie。

— David Flanagan

<http://www.davidflanagan.com>

2005 年 2 月





欢迎来到 Java 的世界！本章一开始将会告诉你 Java 是什么并说明它与其他程序语言有哪些不同的功能；接着就会概述一下这本书的结构，特别会强调在 Java 5.0 中有些什么新的功能；最后，我们还会提供一个简单的 Java 程序范例，你可以将它输入计算机，然后编译并运行它。

## 什么是 Java？

在讨论 Java 时，将 Java 程序语言、Java 虚拟机（Java virtual machine, JVM）以及 Java 平台加以区别是很重要的一件事。Java 程序语言是用来编写 Java 应用程序、applet、servlet 以及 Java 组件等的语言。当 Java 程序被编译时，它会被转换成字节码（byte code），字节码是 CPU 构架（即 JVM）的具有可移植性（portable）的机器语言。JVM 可直接以硬件方式来实现，但通常都是以软件程序的形式来表现，而字节码便是由 JVM 来解释与运行的。

Java 平台是不同于 Java 程序语言和 JVM 的，它是存在于每个 Java 安装系统（Java installation）中的预定义 Java 类（class）集合，而这些 class 可以被所有的 Java 程序所使用。Java 平台有时候被称为 Java 运行环境或是核心 Java API（Application programming interfaces）。Java 平台可以使用其他额外的标准扩展功能（standard extensions），而这些扩展 API 仅存在于某些 Java 安装系统中，并不保证存在于所有的安装系统中。

## Java 程序语言

Java 程序语言是目前最先进的面向对象编程语言，它的语法和 C 语言相当类似。Java 的

设计者致力于让 Java 语言更为强大，同时他们也尝试避免其他面向对象编程语言（如 C++）所有的过分复杂的特点。设计者通过让 Java 语言更为简易好用，使得程序员能写出更强大且无错误（bug-free）的代码。也正因为 Java 的精良设计与完全具备下一代程序语言所必须拥有的所有特点，Java 语言早已受到程序员相当程度的欢迎，尤其是在他们使用过其他不易于使用且功能更小的程序语言之后，他们更深深地认为能使用 Java 语言是一件非常幸福的事。

Java 5.0 是最新版本的 Java 语言（注 1），包括了一些新的程序语言特点，增加了 Java 语言的复杂度与强大性。大部分有经验的 Java 程序员都非常喜欢这个新的特点。

## Java 虚拟机

Java 虚拟机，或称 Java 解释器（interpreter），是 Java 安装系统（installation）最重要的一部分。Java 程序被设计成具可移植性，但也只能在安装了 Java 解释器的平台上运行。Sun 为它们的 Solaris 操作系统与 Microsoft Windows 及 Linux 平台制作了 JVM，还有很多其他的厂商，包括 Apple 及其他与 Unix 相关的公司都为自己的平台提供了 Java 解释器。JVM 不只用于桌面系统，它还被移植到机顶盒（set-top boxes）及使用 Windows CE 及 PalmOS 的手持式设备中。

虽然解释器在一般的概念中被认为是个不具备高性能的系统，但是 JVM 的性能却表现得非常优异，同时还在不断的进步当中。值得特别注意的是一个称做实时（just-in-time, JIT）编译的 JVM 技术，凭借这个技术，Java byte code 可以被转换为本地平台使用的机器语言，从而加快了需要重复运行的程序代码的运行效率。

## Java 平台

Java 平台和 Java 程序语言及 Java 虚拟机一样重要。所有使用 Java 语言编写的程序都必须依赖构成 Java 平台的预定义类集合才能顺利运行（注 2）。Java 类被分为多个不同的包（package），Java 平台的包依功能来定义，如输入/输出、网络、图形、用户界面的创建、安全性以及其他许许多多的功能。

---

注 1：Java 5.0 的版本号码对 Sun 来说是个重大的变革。因为 Java 的前一个版本是 Java 1.4，所以有时候你可能会听到另一个非正式的称呼为 Java 1.5，其实它就是指 Java 5.0。

注 2：类是一个由 Java 程序代码所组成的模块，它定义了数据结构以及操作该数据的一些 method（又称为 procedure、function 或 subroutine）。



了解平台这个名词所代表的意义是非常重要的。对于一个计算机程序员来说,所谓的平台是指当他在写程序时所需使用的且预先被定义好的API,这些API通常都是由目标计算机上的操作系统所定义的。因此,程序员专为Microsoft Windows所写的程序与专为以Unix为基础的操作系统所写的具有相同功能的程序,它们所使用的API必定是不相同的。也就是说,Windows与Unix是不同的平台。

Java并不是个操作系统,不过Java平台所提供的API的广度与深度都比得上现今的操作系统所提供的。Java平台可以让你使用一些较为高级的功能,来编写出适合某个操作系统的应用程序。使用Java平台所编写出的应用程序可以在任何支持Java平台的操作系统上运行,这就意味着你不需要为你的程序分别创建Windows、Macintosh及Unix的版本。一个Java程序便可以在所有不同的操作系统上运行,这也充分地解释了Sun对Java的宣传口号“一次编写,到处运行”。

Java平台虽不是一个操作系统,但是对程序员而言,它提供了另一种程序开发的模式,也因此受到了大家的欢迎。Java平台排除了程序员对操作系统的依赖性,同时凭借写一份程序便可在任何的操作系统上运行的特点,也让终端用户可以自由地挑选自己想使用的操作系统。

## Java 的版本

当编写本书时,Java有六个主要的版本,它们分别为:

### Java 1.0

这是Java第一个正式发行的版本,它包括了8个包(package)与包中的212个类。它非常的简单,不过现在已经完全过时了。

### Java 1.1

此版本的Java平台的包已扩展至23个,其类也扩展至504个。在这个版本里新增了inner class,这对Java来说是个很重要的变革,同时在Java VM的性能上也改进了很多。很遗憾地,这个版本现在也已经过时了。

### Java 1.2

此版本对Java来说是非常重要的一个版本:Java平台的包扩展到了59个,类也扩展到了1520个,同时新增了几个和集合、列表、对象的映射以及用于创建图形用户界面的Swing API相关的API。因为有很多新的功能在1.2的这个版本中被开发出来让程序设计员使用,所以这个平台被重新命名为“Java 2平台”。这个“Java 2”的名称只是单纯的标志而已,和Java实际上发行的版本编号是没有任何相关的。

### Java 1.3

此版本是对 Java 平台做些微小的更新,主要是修正一些错误和 Java 本身的稳定性及性能的改善(包括高性能的“HotSpot”虚拟机)。此版本新加的功能有 Java Naming Directory Interface (JNDI) 与 Java sound API 的改善。同时,在这个版本里最有趣的类就是 `java.util.Timer` 及 `java.lang.reflect.proxy` 了。Java 1.3 版总共有 76 个包及 1842 个类。

### Java 1.4

此版也是一个重大的革新版本,新加了几个重要的功能,同时包也扩展到了 135 个,类增至 2991 个。新增的功能有:提供高性能的低级 I/O API、支持了正则表达式的模式匹配、`loggin` API、参数选择 API、新的 Collections 类、为 JavaBean 提供以 XML 为基础的持续性机制、支持使用 DOM 及 SAX API 分析 XML、支持使用 Java 认证与授权服务 (JAAS) API 的用户认证机制、支持使用 SSL 协议的网络安全连接、支持密码术、读写图像文件的 API、网络打印的 API、一些 Swing API 中新的 GUI 组件及 Swing 中简单的拖放结构。除了以上的更动外,Java 1.4 版本也为 Java 语言引进了断言语句 (assert statement)。

### Java 5.0

此版为 Java 的最新版本,在这个版本中有某些 Java 核心语言的修改,包括了 generic type、enumerated type、annotations、varargs methods、autoboxing 及新的 `for/in` 语句。因为主要语言变动了,所以版本编号就增加了。如果 Sun 之前没有因为商业上的考虑使用“Java 2”这个专有名词的话,这个版本在逻辑上应该被称作“Java 2.0”。除了语言本身的更改外,Java 5.0 同样也将包扩展到了 166 个,同时类也增至 3562 个。特别值得注意的有 concurrent 程序的工具、远程管理的结构及用于远程管理的类与 Java VM 本身的测试工具。

在这本书的前言中的 Java 变更列表里,也提到了在新的 Java 语言与平台中使用指针覆盖的功能。

使用 Java 写程序,你必须使用 JDK,而且 Sun 也为每一个版本的 Java 提供了该版本的 JDK。另外,也请不要将 JDK 与 JRE (Java Runtime Environment) 混淆了:JRE 包括了需要运行 Java 程序的所有事情,但它不包括 Java 程序的开发工具(如编译器)。

除了被大部分的 Java 开发者与本书所使用的标准版本外,Sun 也提供了给较专业的开发者使用的 J2EE (Java 2 Platform, Enterprise Edition) 版本,及供消费电子产品系统(如手持式 PDA 与移动电话)使用的 J2ME (Java 2 Platform, Micro Edition) 版本。若想对 J2EE 及 J2ME 有更多的了解,您可以参考《Java Enterprise in a Nutshell》与《Java Micro Edition in a Nutshell》这两本书籍。



## 使用 Java 的好处

为什么要使用Java呢？学习这样的一个新语言与新平台是有价值的吗？这个章节将会告诉你使用Java到底有那些好处。

### 一次编写，到处运行

Sun认为“一次编写，到处运行”是Java平台最主要的核心价值。以商业术语来说，这句话代表Java技术最重要的承诺是你只要写一次程序（即可被编译为字节码在Java平台上运行），便能在任何地方运行该应用程序。

可在任何地方运行你的应用程序！这就是Java平台所提供的最重要的功能。很幸运地，对Java的支持已经越来越普及了，所有主要的操作系统都已经将Java集成到系统中了，Java也被内置于许多受到欢迎的浏览器里，同时它也正被集成于消费电子产品内，如电视机顶盒、PDA及移动电话等。

### 安全性

使用Java的另一个好处是它的安全性功能，Java语言与平台都是以安全性为基础构建出来的。Java平台允许用户在网络上下载非置信（untrusted）的程序代码并在安全的环境下运行它，因此该程序代码将不会造成任何的伤害。它无法使用病毒来侵害宿主计算机系统，也无法有从硬盘中读取或写入任何文件等其他动作。这样的能力让Java更显出独特性与安全性。

Java 1.2更进一步地提升了Java的安全性，它使得安全级别与限制性具有高度的可配置性，同时将这些功能扩展至applet上。对Java 1.2而言，任何的Java程序代码，不管是applet、servlet、JavaBeans组件，还是一个完整的Java应用程序，都可以在限制访问权限的模式下运行，这样就可以防止其可能会对宿主系统所造成的伤害。

Java语言与平台的安全性漏洞已经由世界各地的专家协助修正过了，这些与安全性有关的漏洞，包括了会造成相当程度伤害的程序错误，都已经被发现且修正过了。因为Java对安全性所作的承诺与保证，所以若现在再发现任何有关于安全性的漏洞都将会是个天大的新闻。到目前为止没有任何一个主流的平台能提供像Java所能够保证的安全性，也没有人敢保证将来都不会有任何Java安全上的漏洞问题，就算Java的安全性还不够完美，但它已被证明强壮到足以解决目前所可能遭遇到的所有威胁，同时也毋庸置疑，它比起其他平台来要好得多了。

## 以网络为中心的程序设计

Sun公司的格言一直都是“网络即是计算机”，Java平台的设计者更深信其重要性，同时把Java平台设计为以网络为中心的模式。从程序员的角度来说，Java能够轻易地通过网络来取得资源，并使用client/server或多层次的结构来创建以网络为基础的应用程序。

## 动态及可扩充的程序

Java既是动态的(dynamic)也是可扩充的(extensible)。Java程序代码是由面向对象的模块所构成的，以这种形式所构成的单位称作类(class)。类分散地存储在不同的文件中，同时只有在需要的时候才会被加载到Java解释器中。这表示应用程序可以在运行的时候决定哪些是需要的类，然后在需要的时候才加载它们。这也表示了程序可以凭借加载它所需要的类来动态地扩充它所具备的功能。

Java平台以网络为中心的设计方式，意味着Java应用程序可以动态地凭借网络加载新的类以扩充它的功能。使用这些功能的应用程序将不再只是一个个的程序代码而已，而是成为了一个相互作用且独立的集合。Java开启了强大的应用程序设计与开发的新纪元。

## 国际化

Java语言与Java平台在一开始发展时便是以全世界为基础观点来加以设计的。Java是目前所被使用的语言中，唯一在核心程序里便具有国际化特点的程序语言，而其他的语言都是采用附加的方式。当大多数的程序语言只使用8位字符来表示英文字母与西欧字母时，Java已使用16位的Unicode字符来表示世界上所有的音标字母与表意文字字符集了。然而Java国际化的特点并不只限于低水平的字符表示，同时还包括了Java平台，这使得使用Java来写国际化的程序时比使用其他环境来得简单得多。

## 性能

正如前面所说的，Java程序被编译成具有可移植性的中间形式，即所谓的字节码，而不是某种机器的专属机器指令，JVM就是解释这些具有可移植性的程序代码来运行Java程序的。这个结构意味着Java程序的运行效率快过使用纯解释式语言所写成的程序或脚本(script)。但一般来说，Java程序的运行速度会比编译成某个特定机器码的C或C++程序的运行速度慢。然而，要特别记得，虽然Java程序被转换为字节码，但并非所有的Java平台都是使用字节码来实现的。为了要提高效率，Java平台在计算的部分具相当的复杂度，如字符串运作的方法都是使用该机器的机器码来实现的。虽然较早的Java版本有性能上的问题，但JVM运行的速度却在新版本中获得了很大程度的改善。JVM经由许多

有效的方法加以调整并最佳化,同时许多实现都包含了实时(JIT)编译器,可以很快地将Java字节码转换成该机器的机器指令。使用复杂精密的JIT编译器,Java程序在运行速度上可以和C或C++所写成的应用程序并驾齐驱。

Java是具有可移植性的解释式语言,Java程序在运行上几乎和不具可移植性的C与C++程序一样快。过去,性能的问题曾经使得程序员不想使用Java,但现在Java在1.2、1.3、1.4与5.0版本中的进步,使得使用性能已不再是个问题了。

## 程序员效率和节省时间

最后,也是最重要的,程序员都非常喜欢使用Java的原因是Java是个优美的编程语言,由许多强大而设计良好的API所组成。程序员们都非常喜欢使用Java来设计程序,同时对程序运行出结果所需的时间感到惊讶。就是因为Java是个简单且设计优美的语言,并拥有设计精巧的API,所以程序员可以写出更好的程序,同时减少程序错误的数目。比起其他平台来说,这可节省很多程序开发的时间。

## Java 程序范例

例1-1是一个计算阶乘的Java程序(注3)。注意,每一行最前面的数字并不是程序的一部分,那只是用来方便我们逐行解释。

例1-1: Factorial.java: 一个用来计算阶乘的程序

```
1 /**
2  * This program computes the factorial of a number  这个程序是用来计算某个数字的阶乘
3  */
4 public class Factorial {                                // 定义一个类
5     public static void main(String[] args) {            // 程序从这里开始
6         int input = Integer.parseInt (args[0]);         // 取得用户输入的数字
7         double result = factorial (input);              // 计算阶乘
8         System.out.println (result);                   // 将结果显示出来
9     }                                                    // main()方法在这里结束
10
11     public static double factorial(int x) {              // 此方法用来计算x!
12         if (x < 0)                                       // 检查输入是否合法
13             return 0.0;                                 // 如果不合法, 则返回0
14         double fact = 1.0;                               // 设定初始值
15         while(x > 1) {                                   // 开始进行循环, 直到 x=1
16             fact = fact * x;                             // 每次都乘以x
17             x = x - 1;                                   // 然后递减x
```

注3: 整数的阶乘是所有小于它的正整数的乘积。例如,4的阶乘可以写成4!,也就是 $4 \times 3 \times 2 \times 1$ ,最后乘积的结果为24。依据定义,0!的值为1。



```
18     }                                // 跳回至循环开始处
19     return fact;                      // 返回结果
20 }                                    // factorial()方法在此结束
21 }                                    // 类在此结束
```

## 编译并运行程序

在我们详看程序如何运行之前，首先我们必须先讨论一下如何去运行程序。为了能编译 (compile) 与运行程序，你需要有一个 JDK 之类的工具。Sun Microsystems 在创造 Java 语言的同时，也为它的 Solaris 操作系统及 Linux 与 Microsoft Windows 平台制作了一个免费的 JDK (注 4)。你可以从 <http://java.sun.com> 下载 Sun JDK 的最新版本。下载后请确认所下载的是 JDK 而不是 JRE，因为 JRE 只能让你运行已经存在的 Java 程序，它不能帮助你编写与编译程序。

Sun JDK 并不是你唯一可以使用的 Java 程序开发环境，有许多的公司都提供了 Java IDE (Integrated Development Environments, 集成开发环境) 以及具高质量的开放源码的 IDE，这些都可以让你从事 Java 程序开发的工作。本书假设你是使用 Sun 的 JDK 与其附加的命令行工具。如果你使用其他公司的产品，请详细阅读产品的说明文件，以便学会如何编译与运行如例 1-1 的程序。

当你安装好 Java 程序开发环境后，接下来的第一件事就是必须将程序输入。请使用你最拿手的文本编辑器将例 1-1 的程序输入 (注 5)，同时请不要输入行 B，因为它们只是方便解说之用的。请注意，Java 语言是有大小写分别的，所以必须格外留意所写字母的大小写，同时你将会发觉很多行都是以分号 (;) 作为结束。忘了输入分号是最常犯的错误，程序若少了它们就会无法运行，所以要特别的小心。你可以忽略掉所有 // 之后的内容，这些是注释，只是为了方便阅读与修改之用，同时它们也会被 Java 忽略。

编写 Java 程序时，你必须使用能够存储文本格式的文本编辑器 (text editor)，而不是使用能提供特殊字形与格式且把文件存储为某些特定格式的文字处理器 (word processor)。我在 Unix 系统上最喜欢使用的是 *Emacs*。如果你使用的是 Windows 系统，同时又没有其他特别喜好使用的程序编辑器，你可以使用记事本或 *WordPad*。如果你使用 IDE，它可能会有一个相当不错的文本编辑器，也请详阅产品说明文件。在输入完程序后，请将

---

注 4： 其他的公司，如 Apple，被允许将 JDK 移植到他们的操作系统中。在 Apple 的例子中，这样的协议会导致在该平台上的最新的 JDK 总是稍晚才会被提供。

注 5： 我建议你最好将这个范例以手动的方式输入计算机，以感受一下 Java 语言的结构。如果你真地不想自己输入的话，当然也可以在 <http://www.oreilly.com/catalog/javanu5/> 这个网址将它下载，同时你也可以在此下载本书中所有的范例。

它存储并命名为 *Factorial.java*。这一点非常的重要，如果你将它存储为其他的名称，该程序将会无法正常运行。

当你将程序代码输入完毕之后，接下来就要编译它了。如果你使用 Sun 的 JDK，则 Java 的编译器是 *javac*。*javac* 是一个命令行工具，所以你只能在终端窗口中使用它，如 Windows 系统的 MS-DOS 或是 Unix 系统的 *xterm*。请于命令行输入以下指令以编译该程序：

```
C:\> javac Factorial.java
```

如果输入这行指令后显示任何错误消息，则可能是在程序输入时犯了一些错误；如果没有任何的错误消息，这就表示编译已经顺利地完成了，此时，*javac* 会创建新的 *Factorial.class* 文件。这就是该程序被编译过后的版本。

在编译完 Java 程序后，你还必须运行它。Java 程序并不会被编译为该机器的机器语言，因此也就不能由该系统来直接运行，而是需要使用所谓的 Java 解释器来运行它。在 Sun 的 JDK 里，解释器是一个命令程序，称作 *java*。如果你要运行刚刚编译好的程序，请输入以下的指令：

```
C:\> java Factorial 4
```

*java* 是用来运行 Java 解释器的指令，*Factorial* 则是我们想要让解释器运行的 Java 程序的名称，而 4 则是输入的数据——也就是我们想要计算阶乘的数字。程序输出结果告诉我们 4! 等于 24。

```
C:\> java Factorial 4
24.0
```

恭喜你，你已经写好、编译并运行一个 Java 程序了。你可以试着计算其他数字的阶乘。

## 解析例 1-1

现在你已经运行过阶乘程序了，我们来好好的逐行解析它，同时来看看 Java 程序的运行原理。

### 注释

程序的前三行是注释，Java 会自动地忽略掉它们，但它们可以给程序员提供有关该程序的一些信息。注释是以 */\** 字符开头，并以 *\*/* 字符做结尾。任何数量的文字，包括多行文字，都可以出现在这两组字符之间。Java 也支持另一种形式的注释，在第 4 行至和第 21

行程序中你可以看到。如果 Java 程序中出现 // 字符, Java 会忽略掉它们以及该行中 // 字符之后的其他字符。

## 定义类

第 4 行是 Java 程序的开始, 它定义了一个名为 `Factorial` 的类, 这也正解释了为什么文件名要取为 `Factorial.java`。文件名指出了该文件包含了一个名为 `Factorial` 类的 Java 源代码。`public` 是个修饰符 (modifier), 它表示该类是公开的且可以给任何人使用。左大括号 { 标明了该类的主体的开头, 该类的主体由此处开始至第 21 行, 同时我们也可以在第 21 行发现右大括号 }。该程序包含了许多成对的大括号, 在这些成对括号内的程序代码形成了一种嵌套式的结构。

类 (class) 是 Java 程序结构的基本单位, 因此在程序的第一行便声明类这一点都不令人感到惊讶。所有的 Java 程序皆为类, 有些程序甚至使用了一个以上的类。Java 是个面向对象的程序语言, 而类正也是面向对象的最基本的结构, 每一个类都定义了独一无二的对象。例 1-1 并不真正是一个面向对象的程序, 所以我们并不打算在这里对对象与类做太深入的讨论, 在第三章中我们会再进行仔细的探讨。现在, 你必须了解的是类定义了一组交互作用的成员 (member), 这些成员有可能是字段 (field)、方法 (method) 甚至是其他的类。`Factorial` 类包含了两个成员, 它们都是 `method`, 我们将在后面的章节对它们做详细的讨论。

## 定义 method

第 5 行定义了 `Factorial` 类的方法 (method)。`method` 在 Java 程序代码中占了相当大的分量。Java 程序可以请求或是调用 (invoke) 一个 `method`, 如果你有编写其他程序语言的经验, 你一定也曾经接触过 `method`, 但是它们通常都被称作函数 (function)、程序 (procedure) 或子程序 (subroutine)。`method` 有参数及返回值, 当你调用 `method` 时, 你可以将你所要操作的数据传递给它, 同时它也会将结果返回给你。`method` 相当类似于代数中常用的函数:

$$y = f(x)$$

在这里, 数学函数  $f$  会对  $x$  所代表的值做某种运算, 同时会将结果返回给  $y$ 。

回到例 1-1 的第 5 行。`public` 与 `static` 这两个关键字都是修饰符 (modifier), `public` 表示该方法是可公开被大家所使用的, 而 `static` 的含义在这里不重要, 在第三章我们将会加以解释说明。`void` 关键字则用来表示方法返回值的形式, 在这个范例中, 它表示此 `method` 并没有返回值。



main是此method的名称，main同时也是一个特殊的名称（注6），当你运行Java解释器时，它会读取你所指定的类，接着会寻找一个叫做main()(注7)的method，当编译器找到此method时，就会开始运行method里面的程序。当main()运行到最后时，该程序也将运行完毕，同时Java解释器也会结束。换句话说，main() method是Java程序的主要进入点，而只把方法名称声明为main()是不够的，方法还必须同时被声明为public static void，如同程序第5行声明的那样。事实上，在程序的第5行，你唯一能修改的就只有args这个词，你可以将它修改成任何你想要使用的词。同时，你也将会在你所有的Java程序中使用到此行，所以请牢牢的把它记住。

main()方法后就是一连串的参数，在这里的main()方法只有一个参数。String[]定义了参数的类型，它是一个字符串数组（一个经过编号的文本字符串），args则是参数的名字。在代数方程式 $f(x)$ 中， $x$ 是用来指一个未知值，args对main() method提供了相同的功能。我们将会看到，args会在method的主体中被使用到，它代表的是某个传递给method的未知值。

正如我所说的，main() method是相当特殊的，当Java解释器开始运行Java类（程序）时，解释器会调用main() method。当你使用以下的方法运行Java解释器时：

```
C:\> java Factorial 4
```

“4”会被传递给main() method以作为args的参数值。更精确地说，一个仅含有“4”的字符串数组会被传递给main()。如果我们使用以下的方式运行该程序：

```
C:\> java Factorial 4 3 2 1
```

那么包含4个字符串“4”、“3”、“2”、“1”的数组会被传递给main() method以作为args的参数值。我们的程序只会使用到数组的第一个字符串，其他的字符串都将会被忽略。

注6： 所有直接经由Java解释器所运行的Java程序都必须拥有一个main() method，且这样的程序常常被称为应用程序(application)。当然，编写一个不直接使用解释器来加以运行但会动态地被加载到其他已经在运行之中的Java程序的程序，也是有可能的。applet就是一个例子，它是被浏览器所运行的程序，而servlet则是被web server所运行的程序。applet在O'Reilly所出版的《Java Foundation Classes in a Nullshell》里有详细讨论，而servlet则在O'Reilly所出版的《Java Enterprise in a Nutshell》里有讨论，在本书我们将只讨论应用程序。

注7： 本书中提到method时，会在method的名称后加上一个小括号()。你也将会发现小括号在method语法上是相当重要的，同时借此我们也能将它与类、字段、变量以及其他的组件加以区分。

最后，在第5行的结尾是一个左大括号，这是 `main()` method 主体的开始，其结尾在第9行的右大括号处。method是由语句 (statement) 所构成的，Java解释器就是依序执行这些语句的。此范例中，第6、7、8行构成了 `main()` method 的主体，每一个语句都是以分号作为区分。这是很重要的Java语法，初学者常常都会忘了这个分号。

## 声明变量与解析输入

在 `main()` method 中的第一个语句，即第6行，声明了一个变量，同时给定了初始值。在任何的程序语言中，变量是一个值的符号名称。在程序中我们已经看到，`args` 是指传递给该 `main()` method 的参数值。method 参数是一种变量，同时 method 也有可能声明额外的“局部”变量 (local variable)，method 可以用局部变量来存储和引用在执行计算时所使用到的中间值。

现在我们要从第6行开始说明了。这行以 `int input` 开始，声明了变量名称 `input`，同时指定该变量为 `int`，也就是 `integer`。Java 可以使用许多种变量类型，包括整数 (`integer`)、实数 (`real`)、浮点数 (`floating-point number`)、字符 (`character`，如字母、数字) 以及字符串。Java 是一个强制类型检查的语言 (`strongly typed language`)，也就是说所有的变量都必须具有一个指定的类型，同时也只有该类型的值才能赋给该变量。我们的 `input` 变量一定是整数，所以不能将其值指定为浮点数或字符串。method 参数也必须指定类型。回想一下，`args` 参数即是一个 `String[]` 的类型。在声明 `input` 变量之后，接着是 `=` 字符，这是Java的赋值运算符 (`operator`)，它是用来设置变量的值。在阅读Java程序代码时，请不要将 `=` 视同于“等于”，请将它读成“被赋值成”。在第二章里我们将看到另一个用来表示“等于”的运算符。

被赋给 `input` 变量的值是 `Integer.parseInt(args[0])`，这是在调用一个 method。`main()` 方法的第一个语句调用了另外一个方法 `Integer.parseInt()`，或许你已经猜到了，这个 method 会“解析”出一个整数。也就是说，它会将一个以字符串表示的数值（如“4”）转换为整数值。`Integer.parseInt()` method 并非Java程序的一部分，而是Java API 最主要的核心部分。每一个Java程序都可以使用那些被定义于核心API里的 class 及 method。

当你在调用一个 method 时，所返回的值（称作自变量，`argument`）会被赋予 method 所指定的参数，而 method 在运行过后也会返回一个值。传递给 `Integer.parseInt()` 的自变量是 `args[0]`，别忘了 `args` 是 `main()` 的参数名称，它被指定为字符串数组。数组内的元素 (`element`) 是按照顺序编号的，且第一号永远都是0，我们只在乎 `args` 数组里的第一个字符串，所以使用 `args[0]` 来访问该字符串。当我们使用这个程序时，第6行的程序代码便会获得我们在命令行中所输入的指令以及在类名之后的第一个字符串

“4”，同时将它传给 `Integer.parseInt()` method。这个 method 会将字符串转换为相对应的整数，并返回该整数值。最后，这个返回的整数值会被赋给 `input` 这个变量。

## 计算结果

第 7 行的语句与第 6 行十分的相似，它声明了一个变量，同时也赋一个值给该变量。被赋给该变量的值是调用了一个 method 后得到的，该变量的名称为 `result`，且为 `double` 类型。`double` 是双精度度（double-precision）浮点数类型，此变量所被赋予的值是由 `factorial()` method 所计算而来的。`factorial()` method 不是标准 Java API 的一部分，它是程序中的第 11 行到第 19 行自行定义的。传给 `factorial()` 的自变量是 `input` 变量在第 6 行经过计算得到的值。后面我们会介绍 `factorial()` method 的主体，或许你从名称上就可以猜到，它会将输入的值计算其阶乘后将结果返回。

## 显示输出

第 8 行调用了名为 `System.out.println()` 的 method，这个经常被使用的 method 是核心 Java API 的一部分，它会让 Java 解释器输出一个值。在这个范例中，输出出来的是 `result` 这个变量被指定的值，也就是计算过阶乘后的结果。如果 `input` 变量的值为 4，那么 `result` 变量的值就是 24 了，这行就会将这个结果输出。

`System.out.println()` method 没有返回值，所以没有变量声明或赋值运算符在这个语句中，也没有任何的值被赋给任何东西。另一种说法，和第 5 行中的 `main()` method 一样，是 `System.out.println()` 被声明为 `void`。

## method 的结束

第 9 行只有一个右大括号 `}`，它表示了 method 的结束。当 Java 解释器运行到这里时，即完成运行 `main()` method，因此便会停止运行。`main()` method 的结尾也正是 `main()` 所声明的 `input` 与 `result` 两个变量及 `args` 参数的变量作用范围（variable scope）的结束。这些变量与参数只有在 `main()` method 中才是有意义的，不能在程序的其他部分被使用，除非程序的其他部分也凑巧声明了相同名称的变量或参数。

## 空行

第 10 行是空行，你可以在程序的任何地方插入空行（blank line）或空格（space），而你可以充分地使用空行来让程序更容易阅读。在这里，空行将 `main()` method 与第 11 行的 `factorial()` method 给分开。你将会发现这个程序会使用许多的空白（whitespace）来将程序缩排。像这样的缩排方法并不是一定必需的，它只是用来表现程序的结构，提高程序的可读性。



## 另一个 method

第 11 行开始定义了被 `main()` method 所使用的 `factorial()` method。和第 5 行相比较，你会发现它们之间有一些相似处与不同处。`factorial()` method 一样有 `public` 与 `static` 这两个修饰符，而它只有一个整数参数 `x`。和 `main()` method 不同的是，`main()` method 没有返回值 (`void`)，`factorial()` 则会返回一个 `double` 类型的值。左大括号表示了 method 的开始，你会发现在程序主体里还有另一对大括号，而第 20 行的右大括号则是 method 的结尾处。`factorial()` method 的主体就和 `main()` method 的主体一样，是由语句组成的，即第 12 行到第 19 行的内容。

## 检查输入的有效性

在 `main()` method 中，我们看到了变量的声明、赋值与 method 的调用，而第 12 行的语句就有点不同了，它是一个 `if` 语句，会执行另一个条件语句。之前我们看过 Java 解释器执行 `main()` method 里的三个语句，它总是以同样的方式、同样的次序来执行这些语句，而 `if` 语句是一种流程控制 (`flow-control`) 语句，它会影响解释器运行程序的方式。

在 `if` 关键字之后的是由一对小括号 `()` 括起来的表达式 (`expression`) 与一个语句。Java 解释器会先求出表达式的值，如果所得的值是真，解释器就会执行这个语句；如果所得的值为假，则解释器会跳过该语句直接执行下个语句。第 12 行的 `if` 语句的表达式为 `x < 0`，解释器会去检查传给 `factorial()` method 的值是否小于 0，如果是的话，则此表达式为真，于是就会执行第 13 行的语句。第 12 行的结尾没有分号是因为第 13 行的语句是 `if` 语句的一部分，只有在语句结尾处才需要加上分号。

第 13 行是一个 `return` 语句，说明了 `factorial()` method 的返回值是 0.0。`return` 也同样是流程控制 (`flow-control`) 语句。当 Java 解释器看到了 `return`，它就会停止运行当前的 method 并立刻返回这个指定的值。`return` 语句可以单独使用，但在这个范例里，`return` 语句是 `if` 语句的一部分，第 13 行的缩排更能显示出这一点 (Java 会忽略掉这些缩排，但缩排对于阅读程序代码确实有相当大的帮助)。只有第 12 行的表达式为真的时候，第 13 行才会被执行到。

在我们要继续下一行之前，应该要来讨论一下为什么第 12 行与第 13 行要放在最前面。如果你尝试着去对一个负的整数计算其阶乘，这将是不正确的，所以这几行是用来确保输入的 `x` 值是合法的。如果该值不合法，那么 `factorial()` 将会返回一个无效的结果 0.0。



## 一个重要的变量

第14行是另一个变量的声明，它声明了`double`类型且名称为`fact`的变量，同时赋给它了一个`1.0`的初始值，此变量的值在我们计算阶乘时将会用到。在Java里，变量可以在任何地方被声明，并没有限制为一定要在`method`或某段程序的开头处声明。

## 循环与计算阶乘

第15行使用了另外一种类型的语句：`while`循环。跟`if`语句一样，`while`语句是由一对小括号所括起来的表达式与一个语句所构成的。当Java解释器看到了`while`语句，它会先求出表达式的值，如果表达式所求出的值为真，则解释器就会去执行后面的语句。解释器会一直重复进行这样的过程，计算表达式里的值，如果表达式的值为真就会执行该语句，直到表达式的值为假为止。第15行的表达式为`x > 1`，所以当参数`x`的值在大于1的情况下`while`语句会重复地被执行。另一种说法是，循环会一直执行到`x`的值小于或等于1时才停止。我们可以从这个表达式得知，如果要将循环停止的话，就必须在循环中修改`x`的值。

与第12及第13行的`if`语句最主要的不同点是，`while`循环是复合语句。所谓复合语句就是指在这一对大括号内是由零个或多个语句所组合而成的。在第15行的`while`关键字之后的是由小括号括起来的表达式，接着是一个左大括号，此循环的主体便是由这里开始，直到第18行的右大括号为止。之前曾经提过，所有的Java语句都是以分号作为结尾。这个规则不适用于复合语句中，你可以发现第18行的结尾没有分号，但复合语句中的语句（第16及第17行）都要使用一个分号结束。

`while`循环的主体是由第16与第17行的语句所构成的，第16行会将`fact`的值与`x`相乘，并将结果存储回`fact`。第17行也相当的类似，它会将`x`的值减1并将结果存回`x`。第16行的`*`字符相当的重要，它是乘法运算符，或许你也已经猜到了，第17行的`-`是减法运算符。运算符（operator）在Java语法中是很重要的，它对一个或两个操作数做运算，同时会产生一个新的值。运算符与操作数组合起来成为一个表达式，就如同`fact * x`或`x - 1`。我们也已经在程序的其他部分看过类似的运算符了，如第15行中使用了大于运算符（`>`）。在表达式`x > 1`中，它会将`x`的值与1做比较，同时这个表达式的值是布尔值（boolean）——非真（true）即假（false），这完全依据比较后的结果。

为了要了解`while`循环的运作，我们可以将自己想象成是Java解释器。假设我们要计算`4!`，在循环开始之前，先让`fact`为`1.0`，`x`为`4`。在执行了一次循环主体后——即在第一个迭代（iteration）之后，`fact`的值为`4.0`，`x`为`3`；在执行了第二次的iteration之后，`fact`为`12.0`，`x`为`2`；在第三次的iteration之后，`fact`为`24.0`，`x`为`1`。当解释器在执行完第三次的iteration之后，它会发现表达式`x > 1`的值不再为真了，因此它会停止执行循环，并开始执行第19行的语句。

## 返回结果

第19行是另一个return语句,和第13行所看到的是一样的,它不会返回像是0.0一样的常量值,它所返回的是fact变量的值。如果传给factorial()函数的x值为4,那么在计算过后fact即为24.0,这就是要返回的值。还记得factorial() method是在程序的第7行被调用的吗?当return语句被执行时,控制权就重回到第7行,而返回来的值就会被指定给result变量。

## 异常

如果你已经逐行仔细地看过例1-1的程序,相信你已经对Java语言的基本原理有了相当的了解(注8)。它是一个简单但却非常重要的程序,演示了Java很多的特点。在这里还要介绍一下Java另一个更为重要的特点,但它在这个程序里并没有出现。你应该还记得该程序在计算阶乘时所用的值是你命令行所指定的数值,如果在运行时没有指定一个数值给它,将会发生什么样的事情呢?

```
C:\> java Factorial
java.lang.ArrayIndexOutOfBoundsException: 0
    at Factorial.main (Factorial.java:6)
C:\>
```

如果你所指定的值不是一个数值,又会发生什么样的事情呢?

```
C:\> java Factorial ten
java.lang.NumberFormatException: ten
    at java.lang.Integer.parseInt (Integer.java)
    at java.lang.Integer.parseInt (Integer.java)
    at Factorial.main (Factorial.java:6)
C:\>
```

上述这两种情形下,都会有错误发生,依照Java术语来说,就是会有一个异常(exception)被抛出(thrown)。当异常被抛出时,Java解释器就会显示出一段信息,用来说明是什么样的异常发生了以及在哪里发生的(上述的两个异常都是发生在第6行)。第一种情形中,异常被抛出的原因是因为在args列表里没有字符串,也就是程序中所使用到的args[0]是不存在的。在第二种情形中,异常被抛出的原因则是因为Integer.parseInt()无法将字符串“ten”转换为数值。在第二章我们将会对异常做更多的介绍,同时也会学到当异常发生时该如何去处理。

注8: 如果你对这个阶乘的程序并没有了解得很透彻,请不要担心,我们将在后面的章节更仔细地介绍Java的语法。然而,如果你对于逐行的解析还是不那么了解的话,那么接下来的章节对你来说可能会有点吃力了。如果是这样的话,你可能要在别处学习Java语言的基础,然后再回到这本书将你所了解的知识给整合起来。Sun的在线Java教学在学习语言时是个非常有用的资源,你可以在<http://java.sun.com/docs/books/tutorial>上取得。



# Java 基本语法

本章将会对 Java 语法做深入的介绍。本书是写给从没用过 Java 程序语言但是至少有使用其他程序语言的经验的读者。当然，对于之前完全都没有任何程序设计经验的读者来说，阅读本章也是相当有帮助的。如果你曾经接触过 Java，本章对你来说会是非常有用的参考文件。在本章中，同时也对 Java 及 C、C++ 语言做了些比较，让程序员们了解 Java 的优点。

本章一开始会从最低级的 Java 语法介绍起，慢慢地会介绍到更高级的结构。本章涵盖的内容有：

- 编写 Java 程序常使用到的字符及这些字符的编码（encoding）。  
直接量值（Literal value）、标识符（identifier）及其他组成 Java 程序的记号（token）。
- Java 可以使用的数据类型。
- 运算符（operator）：在 Java 中将单独的记号加以结合成为表达式。
- 语句（statement）：它是由表达式与其他的语句加以组合而成的 Java 程序代码的逻辑块。
- 方法（method）：method 也被称作函数（function）、程序（procedure）或子程序（subroutine），它是由 Java 语句所构成的，同时也可以被其他的 Java 程序调用。
- 类（class）：它是由 method 与字段（field）所构成的，同时也是构成 Java 程序的最主要元素，并且是面向对象编程的基础。第三章将会非常详细地讨论 class 与 object（对象）。
- 包（package）：由相关的 class 组合而成。



- **Java 程序**: 是由一个或多个相互影响的 class 组合而成, 同时这些 class 可来自一个或多个 package。

大部分的程序语言的语法都是很复杂的, Java 也不例外。一般来说, 在讨论程序语言中的某些元素 (element) 时, 难免都会提到一些尚未讨论的元素。例如, 在介绍 Java 里所支持的运算符与语句时, 都会涉及到对象 (object); 同样地, 在讨论对象的时候, 也都会提到运算符与语句。在学习 Java 或其他语言的过程中, 都会有类似的情形发生, 必须要不断地前后反复学习, 才能将语言学习得更好。如果你刚接触 Java (或是类似 Java 的程序语言), 将本章与下个章节阅读两遍后, 你将会从中获得一些相关的重要概念。

## Java 概述

在我们从最基础的 Java 语法开始学习之前, 先来对 Java 程序做个概述。Java 程序是由一个或多个文本, 或 Java 源代码的编译单元 (compilation unit) 组合而成的。在本章的最后部分, 我们将会说明 Java 文本的结构以及如何编译与运行 Java 程序。每一个编译单元由 package 的声明开始, 接着就是 import 的声明, 不过有时候也有可能没有 import 声明。这些声明赋予了命名空间, 且编译单元会在其中定义名称, 并会从这些命名空间导入名称。我们将在之后的“包与 Java 命名空间”一节中讨论 package 与 import。在 package 与 import 的声明之后就是引用类型的定义, 通常这些是 class 或 interface 的定义, 但在 Java 5.0 以后, 他们也可以是枚举类型或 annotation 的定义。引用类型的一般功能将在本章后半段做说明, 并在第三章与第四章都会深入地介绍各种不同的引用类型。

类型的定义包含的成员有字段 (field)、方法 (method) 与文件结构 (constructor)。method 是最重要的一个成员, 它是由语句组成的 Java 程序代码块。大部分的语句包括了表达式, 它是由运算符与基本数据类型的值组合而成的。最后, 使用关键字编写语句, 用来代表运算符的标点符号以及在程序中出现的 literal value 都是符号, 这些都在稍后做说明。就如同本章节的名称一样, 此章节从最小单元的记号 (token) 开始说明, 再慢慢地渐近至最大的单元。因为这些概念都互为基础, 所以强烈建议读者一定要依照顺序来阅读本章节。

## 词汇 (Lexical) 结构

本章在说明 Java 程序的词汇结构, 将会从编写 Java 程序会使用到的 unicode 字符集开始讨论。含括了组成 Java 程序的记号 (token)、说明用的注释 (comment)、标识符 (identifier)、保留字、直接量 (literal) 等。



## Unicode 字符集

Java 程序是由 Unicode 所编写成的，你可以在 Java 程序的任何地方使用 Unicode 字符集，包括注释及变量名称的定义都可以使用它。不像 7 位的 ASCII 字符集那样只适用于英语，也不像 8 位的 ISO Latin-1 字符集只适用于主要的西欧语言，Unicode 字符集则可以适用于地球上所有的语言。16 位的 Unicode 字符集一般来说都是使用 UTF-8 的编码方法写入文件，它会将 16 位的字符转换成字节流（stream），以这种方式所设计的格式使得纯 ASCII 文本（7 位的 Latin-1）都是合法的 UTF-8 字节流。因此，你可以使用纯 ASCII 编写程序，而它们仍然会被视为是合法的 Unicode。

如果你想要在程序中嵌入一个 Unicode 字符，你可以使用特殊的 Unicode 转义序列（escape sequence）`\uxxxx`，也就是一个反斜线加上一个小写的字母 `u`，后面再接上 4 个 16 进制的字符。例如，`\u0020` 代表的是空格符、`\u03c0` 是字符  $\pi$ 。

在 Java 5.0 以后的版本是使用 Unicode 3.1，包括了需要用 21 位来表示的“增补字符”（supplementary character）。若要以 16 位编码的 Unicode 字符表示增补字符，可以使用 *surrogate pair*，它是由两个有序列的 16 位字符所占的特定区域的编码。如果你需要在 Java 源代码中加入一个或一个以上的增补字符时，可以使用两个 `\u` 序列来表示 *surrogate pair*（对 *surrogate pair* 编码的详细说明并不在本书的讨论范围内）。

## 大小写区分与空白

Java 是一个区分大小写的语言，而关键字一直都是以小写字母来表示。也就是说，`while` 与 `WHILE` 和关键字 `while` 是不同的；同样地，如果你在程序里声明了一个名为 `i` 的变量，你就不能使用 `I` 来引用该变量。

Java 程序会忽略掉空格、跳格（tab）、转行（newline）与其他的空白，除非它和引号字符与字符串直接量（literal）一起出现。程序员一般都会使用空白来编排与缩排他们的程序，为的只是方便阅读，同时你也将在本书的范例程序中看到一般的使用缩排的方法。

## 注释

注释是为了要让程序更容易阅读，而且它会被 Java 编译器忽略。Java 支持三种类型的注释。第一种是单行注释，它是以 `//` 开始，一直到该行的结束。例如：

```
int i = 0;    // 初始设定循环变量
```

第二种是多行注释，它是由 `/*` 字符开始，可以有很多行，直到 `*/` 字符所在行为止，任何在 `/*` 与 `*/` 字符之间的文字都会被 Java 编译器忽略。虽然这类的注释一般都是用得多

行的情况下，但它也可以被使用于单行注释。另外，这类的注释不能嵌套使用（即一组 `/* */` 字符不能出现在另一组之内）。当在使用多行注释时，程序员常会使用额外的 `*` 字符好让注释更加的清楚，这里有个典型的多行注释范例：

```
/*
 * 首先，与服务器建立连接
 * 如果连接失败，立即离开
 */
```

第三种注释是第二种的特例，如果注释是以 `/**` 字符开始，就会被当成是特殊的文档注释。和正常的多行注释一样，文档注释是以 `*/` 做结尾，同时它也不可嵌套使用。当你在写 Java class 时，会希望有其他的程序员使用你所写的 class，此时便可以使用文档注释将关于 class 与 method 的说明文件直接嵌入源代码里。一个名为 javadoc 的程序会提取这些注释并加以处理，同时为你的 class 创建在线说明文件。文档注释可包含 HTML 标记，也可以使用其他 javadoc 所能理解的语法。例如：

```
/**
 * Upload a file to a web server.
 *
 * @param file The file to upload.
 * @return true on success,
 *         false on failure.
 * @author David Flanagan
 */
```

注释可以在 Java 程序的任何记号（token）之间出现，但不能在字符串中出现。一般来说，注释不会在被双引号括住的字符串中出现，如果注释和字符串直接量（literal）一起出现，那么它就是该字符串的一部分了。

## 保留字

以下是 Java 的保留字，它们是程序语言中语法的一部分，所以不能用作变量名称或 class 等。

abstract	const	final	int	public	throw
assert	continue	finally	interface	return	throws
boolean	default	float	long	short	transient
break	do	for	native	static	true
byte	double	goto	new	strictfp	try
case	else	if	null	super	void
catch	enum	implements	package	switch	volatile
char	extends	import	private	synchronized	while
class	false	instanceof	protected	this	

我们将在稍后对这些保留字做介绍。其中有些是基本类型的名称，有些是 Java 语句的名

称，这两种保留字在稍后会做说明。还有其他的是用来定义 class 与 class 的成员（将在第三章中介绍）。

请注意，`const` 与 `goto` 虽然也是保留字，但实际上并不会在 Java 程序中被用到。`strictfp` 是在 Java 1.2 中被加入的，`assert` 是在 Java 1.4 中被加入的，`enum` 则是在 Java 5.0 才被加入的。

## 标识符

标识符 (identifier) 是 Java 程序中所用的一个简单的名称，如 class、class 中的 method 或是 method 中的变量声明。标识符可以是任意的长度且可包含字母与 Unicode 字符集及数字。标识符不可以以数字为开头，因为编译器会将它认为是数值直接量 (literal)。

一般来说，标识符不能包含除了 ASCII 的下划线 (`_`)、美元符号 (`$`) 及 Unicode 货币符号 (如 `£` 与 `¥`) 以外的符号。依据惯例，货币符号都会被保留下来，好让某些程序的解析器自动地产生标识符。在程序代码中应尽量避免标识符使用到这些货币符号，对于会自动产生标识符的冲突问题你可以不用太担心。正式的说法，标识符允许作为开头的字符与标识符内可使用的字符在 `java.lang.Character` class 的 `isJavaIdentifierStart()` 与 `isJavaIdentifierPart()` 中被定义。

以下是合法标识符的范例：

```
i      x1      theCurrentTime      the_current_time      q
```

## 直接量

直接量是在 Java 程序代码中直接出现的值。它们包括了整数、浮点数、单引号括起来的字符、双引号括起来的字符串、保留字 `true`、`false` 与 `null`。以下皆为直接量的范例：

```
1      1.0      '1'      "one"      true      false      null
```

表示数值、字符与字符串直接量的语法，将在后面的“基本数据类型”一节中做说明。

## 标点符号

Java 也使用到了一些标点符号当作符号。Java 语言将这些字符分成分隔符 (separator) 与运算符 (operator) 这两个类型。

属于 separator 的有：

```
()  {  }  [  ]  <  >  :  ;  ,  .  @
```

属于 operator 的有：

```

+      -      *      /      %      &      |      ^      <<      >>      >>>
+=     -=     *=     /=     %=     &=     |=     ^=     <<=     >>=     >>>=
=      ==     !=     <      <=     >      >=
!      ~      &&     ||     ++     --     ?      :
```

我们将在书中看到 separator，同时也会在后面的“表达式与运算符”一节中分别为每个运算符做说明。

## 基本数据类型

Java支持8种基本数据类型，正如表2-1中所列出的。基本数据类型包括了布尔类型、字符类型、4种整数类型及2种浮点类型。4种整数类型与2种浮点类型的差别是用来表示它们的位数不同，因此所能表示出的数字范围也就不相同。下一节会对这些基本数据类型做个概述。除了这些基本数据类型之外，Java也支持了非基本数据类型，如类(class)、接口(interface)与数组(array)，这些复合类型(composite type)，也称作引用类型(reference type)，将在稍后的“引用类型”一节中做介绍。

表 2-1：Java 基本数据类型

类型	包含	默认值	大小	所能表示的范围
boolean	true 或 false	false	1 位	NA
char	Unicode 字符	\u0000	16 位	\u0000 ~ \uFFFF
byte	有符号整数	0	8 位	-128 ~ 127
short	有符号整数	0	16 位	-32768 ~ 32767
int	有符号整数	0	32 位	-2147483648 ~ 2147483647
long	有符号整数	0	64 位	-9223372036854775808 ~ 9223372036854775807
float	IEEE 754 浮点值	0.0	32 位	±1.4E-45 ~ ±3.4028235E+38
double	IEEE 754 浮点值	0.0	64 位	±4.9E-324 ~ ±1.7976931348623157E+308

## 布尔类型

布尔类型是用来表示真实性的值。这个类型只有两种可能的值，用来表示两种布尔状态：开或关、是或否、真或假。Java保留了true与false两个关键字来表示这两种布尔值。

C与C++的程序员应该会发现Java对于布尔类型的限制相当的严格：布尔值不能被转换



为其他数据类型的值。特别是，布尔值不是一个整数类型，所以整数值也不能被转换为布尔类型。换句话说，你不能在 Java 的程序中使用以下的简写：

```
if (o) {
    while(i) {
    }
}
```

Java 会强制你写出一个更为简单易懂的程序代码，也就是你必须更明确地指出你所想运行的比较式：

```
if (o != null) {
    while(i != 0) {
    }
}
```

## 字符类型

字符类型用来表示 Unicode 字符。令许多有经验的程序员感到惊讶的是，Java 字符值的长度为 16 位，而在实际上这样的长度是非常正确的。如果你要在 Java 程序中加入字符直接量 (character literal)，只要将它放入单引号之间即可：

```
char c = 'A';
```

当然，你也可以把任何的 Unicode 字符当成字符直接量，而且你可以使用 Unicode 转义序列 \u。此外，Java 支持了许多其他的转义序列，让你能更方便地使用一些常用但却无法显示的 ASCII 字符，如换行字符等，以及一些在 Java 中具有特殊意义的标点符号字符。例如：

```
char tab = '\t', apostrophe = '\'', nul = '\000', aleph = '\u05D0';
```

表 2-2 列出了在字符直接量中可以被使用的转义字符。这些字符也可以被使用于字符串直接量中，稍后我们也将会对此转义字符加以介绍。

表 2-2: Java 转义字符

转义序列	意义
\b	退格
\t	水平 tab 键
\n	换行
\f	换页 (form feed)
\r	回车 (carriage return)
\"	双引号

表 2-2: Java 转义字符 (续)

转义序列	意义
\'	单引号
\\	反斜线
\xxx	以 xxx 编码的 Latin-1 字符。xxx 是八进制 (base 8) 的数字, 介于 000 到 377 之间  \x 与 \xx 的格式也是合法的, 如 '\0', 但并不建议使用, 因为在字符串常量中会造成混乱, 字符串常量的转义序列只有一个数字
\uxxxx	以 xxxxx 编码的 Unicode 字符, xxxxx 是 4 个十六进制的数字。在 Java 程序里, Unicode 转义字符可以出现在任何地方, 不是只限于字符或字符串直接量而已

字符值可以被转换为整数类型, 同样地, 整数值也可以被转换为字符类型。与 byte、short、int、long 不同的是, 字符是一个无符号 (unsigned) 类型。Character class 定义了一些有用的 static method 来处理字符, 包括 isDigit()、isJavaLetter()、isLowerCase() 与 toUpperCase()。

Java 语言的字符类型实际上就是用 Unicode 所设计的。Unicode 的标准也在逐渐的发展中, Java 每出一次新的版本就会将最新的 Unicode 版本收录在该版本中, Java 1.4 使用了 Unicode 3.0 而 Java 5.0 则收录了 Unicode 4.0。这是非常重要的, 因为 Unicode 3.1 是第一个加入字符 (character) 的编码以及码点 (codepoint) 不是 16 位的版本。这些增补字符 (supplementary character) 大多是不常被使用的汉字 (中文字), 它占了 21 位, 无法使用一个 char 值来表示。你必须使用 int 值来表示增补字符的码点, 或者将它编码为两个 char 值的 “surrogate pair”。除非你在写程序时使用的是亚洲语言, 否则不太可能会使用到任何的增补字符。如果你知道将会处理到无法放入 char 的字符时, Java 5.0 为 Character、String 以及其他相关的 class 新增了一些 method, 好让程序员们可以在文字中使用 int 码点。

## 字符串

除了 char 类型之外, Java 还有个数据类型让你在文字中也能够使用字符串 (通常称为 string)。String 类型是一个 class, 它并不是 Java 语言里的基本数据类型, 因为字符串常常被使用到, 所以 Java 也提供了字符串语法供程序员们在程序中使用。String 直接量的写法是在两个双引号间输入任意的文字, 例如:

```
"Hello, world"  
" 'This' is a string!"
```

字符串直接量可以包含任何的转义序列, 此时转义字符会被当作 char 直接量 (见表

2-2)。使用\"序列可以在字符串中加入双引号。因为String是引用类型，所以string literal会在本章稍后的“对象直接量”一节中被详细地说明，第五章则会告诉你一些Java中的String对象。

## 整数类型

Java的整数类型有byte、short、int、long。如表2-1所示，这四种类型的差别就是位数的不同与每种类型所能表示的数值范围。所有的整数类型都可以表示有符号数(signed number)，Java并没有C与C++里所有的unsigned关键字。

这些类型的写法就正如你所预期的：都是十进制数的字符串所构成的，前面可以加上减法运算符以表示负数（注1）。这里有些合法的整数写法：

```
0
1
123
-42000
```

整数直接量也可以用十六进制或八进制来表示。若是以0x或0X开始就表示是十六进制的数字，其中使用字母A到F（或a到f）来表示10到15。若以0开始则代表了八进制的数字，其中不包含数字8和9。Java并不允许整数直接量使用二进制来加以表示。合法的十六进制与八进制的写法如下：

```
0xff          // 十进制的255，以十六进制来表示
0377          // 十进制的255，以八进制来表示
0xCAFEBAFE   // 一个用来标识Java class文件的奇特数字
```

整数直接量是32位的int值，但若在最后加上L或l，它们就会变成64位的long值：

```
1234          // 一个int值
1234L         // 一个long值
0xffL        // 另一个long值
```

Java中的整数计算是取模的(modular)，也就是说，当你超过了该整数类型所能表达的范围时，在计算中不会产生任何的overflow或underflow，它会把运算的结果回绕(wrap)后返回，例如：

注1：严格地说，负号是在literal上操作的运算符，但并不是literal的一部分。同样地，所有的整数literal都是32位的int值，除非该整数的后面加了一个字母L，在这样的情况下，它们就是64位的long值。在这儿并没有byte或short literal的特别语法，但int literal通常都会依需要被转换成这些较短的类型。例如，在以下的程序代码中：

```
byte b = 123;
```

123是一个32位的int literal，它在赋值语句中会被自动地转换(不需要强制转换)为byte。

```
byte b1 = 127, b2 = 1;           // 最大的字节是 127
byte sum = (byte)(b1 + b2);       // 总和会回绕成 -128, 它是最小的字节
```

不管是 Java 编译器还是 Java 解释器, 在发生这种情形时都不会警告你。当在做整数计算时, 你必须确保所使用的类型有足够的表示范围。整数除以零以及求得任何数除以零的余数都是不合法的, 这样会造成抛出 `ArithmeticException` 异常。

每一个整数类型都有一个相对应的 wrapper class: `Byte`、`Short`、`Integer` 与 `Long`。这些 class 都会定义 `MIN_VALUE` 与 `MAX_VALUE` 两个常量, 而这两个常量用来描述该类型的范围。这些 class 也定义了一些有用的静态方法 (static method), 如 `Byte.parseByte()` 与 `Integer.parseInt()`, 用来将 string 转换为 integer 值。

## 浮点类型

在 Java 中是以 `float` 与 `double` 数据类型来表示实数 (real number)。如表 2-1 所示, `float` 是 32 位的单精度浮点数, `double` 是 64 位的双精度浮点数。这两种类型都遵循 IEEE 754-1985 的标准, 而此标准也定义了浮点数的格式与浮点数之间的运算方式。

Java 程序中的浮点数可以是一个数字字符串后面加一个小数点, 最后再加一个数字字符串, 这里有些范例:

```
123.45
0.0
.01
```

浮点数直接量也可以使用指数 (或称科学计数法) 来加以表示, 即在数字的后面紧接着字母 `e` 或 `E` (表示指数) 以及另一个数字。第二个数字所表示的是以 10 为底数的幂, 整个表达式所代表的数字是第一个数字乘以第二个数字所得的数值, 例如:

```
1.2345E02      // 1.2345 * 102 或 123.45
1e-6           // 1 * 10-6 或 0.000001
6.02e23        // 阿伏加德罗常量: 6.02 * 1023
```

浮点数直接量在默认的情况下是使用 `double` 值, 若要使用 `float` 类型, 则要在数值后面加上 `f` 或 `F`:

```
double d = 6.02E23;
float f = 6.02e23f;
```

浮点数直接量不能够被表示为十六进制或八进制。

对大部分的实数来说, 因为其本质的关系, 无法精确地使用有限的位数来加以表示。因此, `float` 与 `double` 值使用这些实数的近似值来加以表示。`float` 是 32 位的近似值,



它至少拥有6个有效十进制数字 (significant decimal digits); double 是64位的近似值, 它至少拥有15个有效十进制数字。实际上, 这两个数据类型对大多数实数间的运算是相当适合的。

除了用来表示一般的数字之外, float 与 double 类型也可以用来表示四种特殊的数值: 正无穷大、负无穷大、零与NaN。当浮点数的计算结果超出 (overflow) float 与 double 所能表示的范围时, 就会出现无穷大的数值; 当浮点数的计算结果小于 (underflow) float 与 double 所能表示的范围时, 就会出现零值。Java 浮点类型有所谓的正零与负零的差别, 这就要看 underflow 所发生的方向来定, 实际上, 正零与负零的运作方式非常相似。最后一个特别的浮点值就是 NaN, 它代表的是“非数字 (not-a-number)”。当进行不合法的浮点数运算时, 如  $0.0/0.0$  这种情况下, 就会出现 NaN 的结果。以下是这几个特殊值的例子:

```
double inf = 1.0/0.0;           // 正无穷大
double neginf = -1.0/0.0;        // 负无穷大
double negzero = -1.0/inf;       // 负零
double NaN = 0.0/0.0;           // NaN
```

就因为 Java 的浮点数可以将 overflow 的值处理为无穷大, 将 underflow 的值处理为零, 并拥有一个特殊的 NaN 值, 所以浮点数的运算永远都不会抛出任何的异常。即使是执行了一个不合法的运算, 如除以零或对负数开根号, 也不会产生异常。

float 与 double 基本数据类型有其对应的 class, 分别为 Float 与 Double, 这两个类都定义了以下几个有用的常量: MIN\_VALUE、MAX\_VALUE、NEGATIVE\_INFINITY、POSITIVE\_INFINITY、NaN。

浮点数的无穷大值的运作方式正如你所预期的, 对任何无穷大的值做加减法都还是无穷大的值。负零的运作方式和正零的运作方法几乎是相同的, 而事实上, 运算符 “==” 会告诉你正零和负零是相同的。有一个方式可以区分正零与负零: 当你将 1.0 除以正零时, 你会得到正无穷大的值; 但若将 1.0 除以负零时, 你则会得到负无穷大的值。最后, 因为 NaN 是一个非数字的值, 所以运算符 “==” 告诉我们它无法等于任何其他数字, 包括它自己在内。若要检查 float 或 double 值是否为 NaN, 你必须使用 Float.isNaN() 与 Double.isNaN()。

## 基本类型转换

Java 允许整数值与浮点值之间做转换, 此外, 每一个字符在 Unicode 编码里都会对应到某个数字, 所以 char 类型也可以与整型与浮点类型间互做类型转换。实际上, 在 Java 中, boolean 值是唯一无法和其他数据类型互做转换的数据类型。

有两个基本的转换方式，一个为放大转换（widening conversion），发生在一种类型的值转换成另一种较大类型的值时——此类型所能表示的数值范围更大。Java会自动执行放大转换，例如，将int值赋给double类型的变量或将char值赋给int类型的变量。

另一个为缩小转换（narrowing conversion），也就是将某一个类型的值转换成另一个较少位数的类型。缩小转换并不是安全的，将整数值13转换成byte类型的数值是相当合理的，但如果将13000转换成byte类型则是不合理的，因为byte类型只能存放-128到127之间的整数值。所以你会发现，不正确地使用缩小转换会发生数据丢失的问题，所以当你尝试使用缩小转换时，Java编辑器会发出警告，即使将被转换的数值在转换为指定的数据类型后仍能维持正确的数值：

```
int i = 13;
byte b = i;    // 编辑器不允许你这样做
```

此规则有一个异常，就是你可以将整数值（int值）赋给byte或short变量，但前提必须是该数值不能超过指定类型所能表示的范围。

如果你需要执行缩小转换，同时你也能确保在转换过后数据不会遗失与失去精度，则你可以强制Java使用强制转换（cast）的语言结构去执行缩小转换。cast的用法就是将所要转换成的数据类型用小括号括起，并放在该数值的前面，例如：

```
int i = 13;
byte b = (byte) i;    // 强制将int转换为byte
i = (int) 13.456;    // 强制将double值转换为int 13
```

基本类型的强制转换最常被用来将浮点数转换为整数。当你如此做时，浮点数的小数部分会被截掉（即浮点数的小数部分会被归零，而不是以最近似的整数值替代）。Math.round()、Math.floor()、Math.ceil()则是能将小数去除的method。

char类型大部分的运作方式和integer类型相似，所以char值可被当成int或long值来使用。因char类型是无符号（unsigned）的，所以它的运作方式和short类型不同，即使两者都是16位宽：

```
short s = (short) 0xffff; // 这些位表示数字 -1
char c = '\uffff';        // 同样的位，表示一个Unicode字符
int i1 = s;                // 将short转换为int得到 -1的结果
int i2 = c;                // 将char转换为int得到 65535的结果
```

表2-3显示了某个基本数据类型所能转换成的类型以及该转换动作是如何被执行的。字母N表示该转换动作是不被允许的；字母Y则表示为放大转换，因此Java会自动地执行；字母C表示该转换为缩小转换，且需要使用强制转换；最后，Y\*表示该转换为自动地放大转换，但在转换的过程中，较低位的几个有效数字可能会遗失，像这样的情形会

发生在 `int` 或 `long` 转换为 `float` 或 `double` 类型时。浮点类型所能表示的范围大于整数类型，所以任何的 `int` 或 `long` 值都可以使用 `float` 或 `double` 来表示。然而，浮点类型都只是该数值的近似值，且其有效数字也没有整数类型来得多。

表 2-3: Java 基本数据类型转换

基本数据类型	转换为:							
	Boolean	byte	short	char	int	long	float	double
boolean	-	N	N	N	N	N	N	N
byte	N	-	Y	C	Y	Y	Y	Y
short	N	C	-	C	Y	Y	Y	Y
char	N	C	C	-	Y	Y	Y	Y
int	N	C	C	C	-	Y	Y*	Y
long	N	C	C	C	C	-	Y*	Y*
float	N	C	C	C	C	C	-	Y
double	N	C	C	C	C	C	C	-

## 表达式与运算符

到目前为止，我们已经学习到了 Java 程序所能使用的基本数据类型以及在 Java 程序中如何加入各种数据类型的直接量，我们也已经学会使用变量来作为某个值的符号名称。这些直接量与变量在 Java 程序中占了相当重要的地位。

表达式 (expression) 是 Java 程序中更高一层的结构，Java 解释器会求出表达式的值。最简单的表达式称作基本表达式 (primary expression)，它是由直接量与变量所组成的。这里有几个表达式的例子：

```
1.7      // 一个浮点数直接量
true     // 一个布尔直接量
sum      // 一个变量
```

当 Java 解释器在计算一个直接量表达式时，所得的结果就是直接量本身；当 Java 解释器在计算一个变量表达式时，所得的结果就是存于该变量的值。

基本表达式还不够有趣，最有意思的是使用运算符将许多基本的表达式组合成更复杂的表达式。例如，以下就是用赋值运算符来结合两个基本表达式——一个变量和一个浮点直接量而成的表达式：

```
sum = 1.7
```



运算符并不是只能够与基本表达式一起使用，它们也可以在更复杂的表达式中被使用。以下皆为合法的表达式：

```
sum = 1 + 2 + 3*1.2 + (4 + 8)/3.0
sum/Math.sqrt(3.0 * 1.234)
(int)(sum + 33)
```

## 运算符总结

你在编程语言中所能使用的表达式完全是以你所能使用的运算符来决定的。表2-4列出了Java提供的所有运算符。表中的P与A栏是用来表示每一组相关运算符的优先级(precedence)与结合性(associativity)。接下来的章节会对这些运算符做更详细的介绍。

表2-4: Java 运算符

P	A	运算符	操作数类型	可执行的运算
15	L	.	对象，成员	对象成员的访问
		[ ]	数组，int	数组元素的访问
		( args )	方法，自变量列表	方法的调用
		++, --	变量	后增加、后递减
14	R	++, --	变量	前增加、前递减
		+, -	数字	正号、负号
		~	整数	bitwise 补码
		!	布尔	布尔 NOT
13	R	new	类，自变量列表	对象创建
		(type)	类型，任何类型	强制转换（类型转换）
12	L	*, /, %	数字，数字	乘法、除法、求余数
11	L	+, -	数字，数字	加法、减法
		+	字符串，任何类型	字符串连接
10	L	<<	整数，整数	左移
		>>	整数，整数	带有符号扩展 (sign extension) 的右移
		>>>	整数，整数	带有零扩展 (zero extension) 的右移
9	L	<, <=	数字，数字	小于、小于或等于
		>, >=	数字，数字	大于、大于或等于
		instanceof	引用类型，类型	类型比较
8	L	==	基本类型，基本类型	等于（拥有相同的值）
		!=	基本类型，基本类型	不等于（拥有不同的值）
		==	引用类型，引用类型	等于（引用同样的对象）
		!=	引用类型，引用类型	不等于（引用不同的对象）

表 2-4: Java 运算符 (续)

P	A	运算符	操作数类型	可执行的运算
7	L	&	整数, 整数	bitwise AND
		&	布尔, 布尔	boolean AND
6	L	^	整数, 整数	bitwise XOR
		^	布尔, 布尔	boolean XOR
5	L		整数, 整数	bitwise OR
			布尔, 布尔	boolean OR
4	L	&&	布尔, 布尔	条件 AND
3	L		布尔, 布尔	条件 OR
2	R	?:	布尔, 任意类型	条件运算符
1	R	=	变量, 任意类型	赋值运算符
		*=, /=, %=,	变量, 任意类型	具有运算功能的赋值运算符
		+=, -=, <<=,		
		>>=, >>>=,		
		&=, ^=,  =		

## 优先级

表 2-4 中的 P 栏定义了每一个运算符的优先等级。优先级定义了运算符被执行的先后顺序, 请考虑如下的表达式:

```
a + b * c
```

乘法运算符比加法运算符有更高的优先级, 所以 a 会与 b \* c 的结果相加。可以将优先级想象成是运算符和操作数关系的紧密度的一个标准, 优先级的数字愈高, 表示运算符与操作数的关系愈紧密。

使用小括号可以改写 (override) 运算符的默认优先权, 并可以明确地指定操作数的执行顺序。上述的表达式可以被改写, 使得加法运算符在乘法运算符之前被执行:

```
(a + b) * c
```

Java 运算符的优先级和 C 语言是兼容的, C 语言的设计者对运算符的优先级的设定方式和 Java 的设定方式是一样的, 使得大部分表达式都不需使用到小括号, 只有少部分的 Java 程序代码必须使用小括号, 如以下的几种情况:

```
// 类强制转换与成员访问结合
((Integer) o).intValue();
```

```
// 赋值与比较结合
while((line = in.readLine()) != null) { ... }

// Bitwise 运算符与比较结合
if ((flags & (PUBLIC | PROTECTED)) != 0) { ... }
```

## 结合性

当表达式中使用了多个具相同优先级的运算符时，运算符的结合性就会支配运算符的执行顺序。大部分的运算符都是具有由左而右的结合性 (associative)，也就是说运算符会按由左而右的顺序来执行。然而，赋值与一元运算符则是具有由右而左的结合性，表 2-4 的 A 栏表示了每一个运算符之间的结合性，L 代表着由左而右，R 则代表着由右而左。

加法与减法运算符皆为由左向右结合的，所以  $a+b-c$  会由左向右被计算：

```
(a+b)-c
```

一元运算符与赋值运算符则会由右向左进行计算，请看以下的复杂表达式：

```
a = b += c = --d
```

它的计算方式为：

```
a = (b += (c = --d))
```

在处理运算符的优先顺序时，运算符的结合性建立了表达式在计算上的默认顺序。这个默认的顺序可以通过小括号而改变，然而，Java 中的默认运算符结合性在设计时就遵循了一般表达式的语法，因此你应该不太需要使用小括号来改变它。

## 操作数的数目与类型

表 2-4 的第四栏定义了该运算符的操作数数目与类型。有些运算符只需要一个操作数，我们称它为一元运算符 (unary operators)。例如，一元减法运算符会改变某个数值的符号：

```
-n // 一元减法运算符
```

然而，大部分的运算符都是二元 (binary) 运算符，必须有两个操作数一起运作。- 运算符其实有两种运作方式：

```
a - b // 减法运算符也是一个二元的运算符
```

Java 也定义了一个三元运算符，通常称为条件运算符 (conditional operator)。它类似于表达式里的 `if` 语句，它的三个操作数是由问号与冒号所区分开的。第二个与第三个操作数都必须转换成相同的类型，该表达式才会被执行：



`x > y ? x : y` // 三元运算符，计算 `x` 和 `y` 何者较大

除了对所能运行的操作数数目有所限制外，对于每个运算符所能计算的操作数类型也是有限制的。表 2-4 中的第四栏列出了操作数的类型，该栏的某些代码需要作更进一步的说明：

### 数字

可以为整数、浮点数或字符（即除了布尔以外的所有基本数据类型）。在 Java 5.0 以后的版本，autounboxing（参考之后的“Boxing and Unboxing 转换”一节），也就是这些类型的 wrapper class（如 `Character`、`Integer`、`Double`），都可以和基本数据类型一样地被使用。

### 整数

可以为 `byte`、`short`、`int`、`long` 或 `char` 值（`long` 值不能被数组访问运算符 `[]` 所使用），与其相对应的对象类型 `Byte`、`Short`、`Integer`、`Long` 与 `Character` 也都是允许的值。

### 引用

对象或数组。

### 变量

可以为变量或其他的符号名称，如数组元素，只要是可以被指定的数值都可以。

## 返回值

就像每一个运算符的操作数都必须有其特定的类型一样，运算结果所产生的值也必须为特定的类型。算术、递增与递减、bitwise 以及移位（shift）运算符，如果有一个以上的 `double` 类型操作数，则其返回值就会是 `double` 类型的值；如果有一个以上的 `float` 类型操作数，则返回值便会是 `float` 类型的值；如果有一个以上的 `long` 类型操作数，则返回值便为 `long` 类型；否则，即使它们的两个操作数都是 `byte`、`short` 或 `char` 类型，它们的返回值也会是 `int` 类型的值，因为这些类型的操作数的位数都小于 `int`。

比较、相等以及布尔运算符都会返回布尔类型的值。每个赋值运算符都会返回它们所赋予的值，而此返回值会与表达式左边的变量类型兼容。条件运算符则会返回其第二个或第三个参数的值（它们必须为同一个数据类型）。

## 副作用

每一个运算符都要计算一个或多个操作数的值，然而，有些运算符除了执行其基本的计算之外，还会有副作用。如果表达式具有副作用，则在计算出表达式的值后将会改变 Java 程序的状态，使得再次计算该表达式的值将会得到不同的结果。例如，++ 递增运算符在

增加变量值时便具有副作用，表达式 `++a` 会递加变量 `a` 的值（也就是将变量 `a` 加 1）且会将结果返回给 `a`。如果再执行一次这个表达式，则返回值的结果将会不同。赋值运算符也具有副作用，例如，表达式 `a*=2` 也可以被写成 `a=a*2`，此表达式所得的值是 `a*2`，但是结果又被返回给了 `a`，因此它也具有副作用。如果所调用的 `method` 具有副作用的话，那么该 `method` 调用运算符 `()` 也具有副作用。有些 `method`，如 `Math.sqrt()`，会计算并返回不产生任何副作用的结果。不过，就一般而言，`method` 实际上是有副作用的。最后，`new` 运算符在创建新的对象时也具有副作用。

## 计算的顺序

当 Java 解释器计算一个表达式时，它会依照表达式中的小括号、运算符优先级以及运算符的关联关系来执行不同的操作。在任何的运算被执行之前，解释器会先求出该运算符的操作数（`&&`、`||` 和 `?` 这几个运算符是异常，这些运算符并不是每次都必须求出它们所有操作数的值）。解释器一定是按由左至右的顺序来计算操作数。当操作数有副作用时，这样的顺序就很重要了。请看以下的范例：

```
int a = 2;
int v = ++a + ++a * ++a;
```

虽然乘法会在加法之前被执行，但 `+` 运算符会先被计算出来，因此，此表达式是用来计算 `3+4*5`，其值为 23。

## 算术运算符

因为大部分的程序主要都是以数值来作运算，所以算术运算符是最常被使用到的运算符。算术运算符可与整数、浮点数甚至是字符一起使用（即可以与除了布尔类型以外的其他基本类型的操作数一起使用）。如果操作数当中有浮点数，则浮点运算便会被使用到；否则，就使用整数运算。这是很重要的，因为整数运算与浮点数运算对于除法的运算及处理 `underflow` 与 `overflow` 的方法是不同的。

算术运算符有：

### 加法 (+)

`+` 运算符会将两个数字相加，`+` 运算符也可以作字符串连接之用。如果 `+` 的任一操作数为字符串，则另一个操作数也会被转换为字符串。当你想要同时使用加法与字符串连接功能时，记得要在适当的地方加上小括号。例如：

```
System.out.println("Total: " + 3 + 4);    // 显示 "Total: 34"，而非 7
```

### 减法 (-)

当 `-` 被当作二元运算符使用时，它会将第一个操作数减去第二个操作数。例如：  
`7-3` 求出的值为 4。`-` 运算符也可以当作负号（一元运算符）使用。

### 乘法 (\*)

\* 运算符会将它的两个操作数相乘。例如：7\*3 求出的值为 21。

### 除法 (/)

/ 操作数会将它的第一个操作数除以第二个操作数。如果两个操作数都是整数，则所得的结果也会是一个整数，而余数会被忽略；若有一个操作数为浮点数，则所得的结果就会为浮点数。当执行整数除以零的动作时，ArithmeticException 异常会被抛出。对浮点计算来说，除以零会得到一个无穷大的值或 NaN：

```
7/3           // 结果为 2
7/3.0f        // 结果为 2.333333f
7/0           // 抛出一个 ArithmeticException
7/0.0         // 结果为正无穷大
0.0/0.0       // 结果为 NaN
```

### 求模 (%)

% 运算符会求出第一个操作数以第二个操作数为模的结果。例如：7 % 3 所得的值为 1，所得值的符号会与第一个操作数的符号相同。虽然 % 运算符一般来说都是与整数操作数一起使用，但它也可以与浮点数一起使用。例如：4.3 % 2.1 求得的值为 0.1。当与整数一起运算，而你想要对零进行求模计算时，将会抛出 ArithmeticException 异常。当与浮点数一起运算时，任何数对 0.0 求模都会得到 NaN，就跟无穷大的值对任何数求模一样。

### 负号 (-)

当 - 被当作负号运算符时，它必须位于某个操作数的前面，也就是说，它可以将某个正数转换为负数或将负数转换为正数。

## 字符串连接运算符

除了数值进行加法运算外，+ 运算符（及相关的 += 运算符）也可以用来连接字符串。如果 + 的某个操作数是字符串，则运算符会将另一个操作数转换为字符串。例如：

```
System.out.println("Quotient: " + 7/3.0f); // 输出 "Quotient: 2.3333333"
```

你必须很小心地在加法表达式的前后加上小括号，如此才能将加法运算符与字符串结合在一起。如果你没有这样做的话，加法运算符就会被当作是字符串连接运算符来使用。

Java 解释器对所有的基本数据类型都内置了字符串转换的功能，可以使用它的 toString() method 来将对象转换为字符串。大部分的 class 都定义了 toString() method，通过此 method，那种类型的对象就可以很容易地使用这种方式被转换为字符串。数组也是利用内置的 toString() method 来被转换为字符串的，但它却无法返回一个表示这个数组内容的有用字符串。



## 递增与递减运算符

`++` 运算符会将它唯一的操作数加1, 该操作数必须是一个变量、数组的一个元素或是对象里的一个字段。此运算符的行为则是由它与操作数的相对位置来决定。当它放在操作数的前面时即所谓的前缀递增 (pre-increment) 运算符, 它会递增该操作数然后求出该操作数的值; 当它放在操作数的后面时, 即所谓的后缀递增 (post-increment) 运算符, 它会递加该操作数, 但在递加动作之前, 它会先计算出该操作数的值。

例如, 以下的程序代码将 `i` 与 `j` 皆设定为 2:

```
i = 1;  
j = ++i;
```

但下面两行代码将 `i` 设定为 2, `j` 设定为 1:

```
i = 1;  
j = i++;
```

相同地, `--` 运算符会将它唯一的操作数减 1, 而该操作数必须是一个变量、数组的一个元素或是对象的一个字段。和 `++` 运算符相同的是, `--` 运算符的运行方式也是由它与操作数的相对位置来决定的。当它被放在操作数的前面时, 它会将操作数递减, 然后返回递减后的值; 当它被放在操作数的后面时, 仍会将该操作数递减, 但返回的是递减之前的值。

表达式 `x++` 与 `x--` 分别与 `x=x+1` 和 `x=x-1` 相同, 除了在使用递加或递减操作数时, `x` 只会被计算一次。如果 `x` 本身即是个具有副作用的表达式, 此时的结果便会有显著的不同。例如, 以下的两个表达式就不相等了:

```
a[i++]++;           // 递加数组的某个元素  
a[i++] = a[i++] + 1; // 将数组的某个元素加一, 并将结果存储于另一个数组元素
```

这些运算符, 不管是放在操作数之前或之后, 都常被用来作为控制循环的递加或递减计数器。

## 比较运算符

比较运算符包括了相等运算符 (equality operator), 用来测试等式或不等式的值, 以及关系运算符 (relational operator), 用来比较具有顺序特性类型 (数值与字符) 数据间的大小关系。这两种类型的运算符所产生的结果为布尔值, 因此它们通常都被用于 `if` 语句与 `while` 及 `for` 循环中, 以作为分支 (branch) 与循环判断之用。例如:

```
if (o != null) ...;           // 不等运算符  
while(i < a.length) ...;      // 小于运算符
```

Java 提供了以下两种等式运算符：

#### 相等 (==)

== 运算符在两个运算符相等的情况下，会返回 `true` 值；如果在不相等的情况下，则会返回 `false` 值。跟基本数据类型一起使用时，它会测试操作数的值是否相等。但对引用类型来说，它会测试操作数是否访问同样的对象或数组。换句话说，它并不会测试两个不同对象或数组是否相同。因此，你无法使用它来测试两个字符串是否相同。

如果 == 被用来比较两个不同类型的数值或字符运算符时，则在比较之前，位数使用得较少的操作数会先转换成较宽操作数的类型。例如：当比较 `short` 与 `float` 时，`short` 会在比较之前被转换为 `float`。对浮点数来说，特殊的情况是负零值的测试结果会等于一般零值或正零值。另外，特殊的 `NaN` 值则不等于任何其他的值，当然也不等于它自身。如果你要测试一个浮点数是否为 `NaN`，可以使用 `Float.isNaN()` 或 `Double.isNaN()` method。

#### 不相等 (!=)

!= 运算符刚好与 == 运算符相反，如果它所测试的两个基本类型的操作数不相同或是两个引用类型的操作数访问不同的对象或数组时，它会返回 `true`，否则就会返回 `false`。

关系运算符可以和数值及字符一起使用，但不能与布尔值、对象或数组一起使用，因为这些数据类型并不是规则的。Java 提供了以下几种关系运算符：

#### 小于 (<)

如果第一个操作数小于第二个操作数，就返回 `true`。

#### 小于或等于 (<=)

如果第一个操作数小于或等于第二个操作数，就返回 `true`。

#### 大于 (>)

如果第一个操作数大于第二个操作数，就返回 `true`。

#### 大于或等于 (>=)

如果第一个操作数大于或等于第二个操作数，就返回 `true`。

## 布尔运算符

正如刚刚我们所看到的，比较运算符会比较它们的操作数并产生布尔值，且比较运算符也常被使用在分支与循环语句中。为了让分支与循环判断所依据的表达式更加地有趣，你可以使用布尔（或逻辑）运算符将多个比较表达式结合成一个较为复杂的表达式。布尔运算符的操作数必须为布尔值，同时它们所产生的值也是布尔值。布尔运算符包括了：

### 条件 AND (&&)

此运算符会对它的操作数执行布尔 AND 运算，只有在它的两个操作数都为 true 时，所得的值才会为 true。如果操作数当中有一个或两个都为 false 时，则所得的值就会是 false。例如：

```
if (x < 10 && y > 3) ... // 如果两个比较皆为 true
```

此运算符（以及所有的布尔运算符，但 ! 运算符除外）的优先级低于比较运算符，因此，这样的例子是完全合法的。然而，有些程序员比较喜欢使用小括号来加以区分，让计算的顺序更明确也表示出来：

```
if ((x < 10) && (y > 3)) ...
```

而你应该使用你认为最容易阅读的方式来编写程序代码。

此运算符被称为条件 AND，是因为它会视情况来计算它的第二个操作数，如果第一个操作数计算结果为 false，则表达式的结果必定是 false，就不会去在意第二个操作数的值。因此，为了增加效率，Java 解释器会抄捷径而忽略掉第二个操作数。因为第二个操作数是不会被计算出来的，所以当你在使用这个运算符计算具有副作用的表达式时必须特别的小心。另外，此运算符会依状况来执行的特性让我们能够写出如下的 Java 表达式：

```
if (data != null && i < data.length && data[i] != -1)
    ...
```

如果第一个或第二个比较式求得的结果为 false 时，则此表达式中的第二个与第三个比较会产生错误。幸运的是，我们并不需要去担心这样的问题，因为 && 运算符有依状况来执行的特性。

### 条件 OR (||)

此运算符会对它的两个布尔操作数执行布尔 OR 操作，如果它的两个操作数中有任何一个值为 true 时，则所求得的结果便为 true；如果两个操作数皆为 false，则结果为 false。如同 && 运算符一样，|| 并不一定会计算它的第二个操作数。如果第一个操作数求得的值为 true，则表达式的结果就会是 true，而不需去考虑第二个操作数的值。因此，此运算符会直接忽略掉第二个操作数。

### 布尔 NOT (!)

此一元运算符会改变它的操作数的布尔值。如果使用在 true 值上，所得的结果会是 false；如果使用在 false 值上，所得的结果就会是 true。它在以下的表达式中特别有用：

```
if (!found) ... // found 是一个在别处声明的布尔变量
while (!c.isEmpty()) ... // isEmpty() method 返回一个布尔值
```

因为 ! 是一个一元运算符，它拥有较高优先级，同时常常必须与小括号一起搭配着使用：



```
if (!(x > y && y > z))
```

### 布尔 AND (&)

当它和布尔操作数一起使用时，& 运算符的运算方式和 && 运算符相同，但它一定会求出两个操作数的值，不管第一个操作数所求得的价值为何。此运算符几乎总是被用作 bitwise 运算符，并与整数操作数一起使用。可能还有很多 Java 程序员不知道它能够与布尔操作数一起使用。

### 布尔 OR (|)

此运算符会对它的两个布尔操作数执行布尔 OR 操作。它和 || 运算符的行为一样，但它一定会求出两个操作数的值，即使第一个操作数的值为 true。| 运算符几乎都被用作 bitwise 运算符并与整数操作数一起使用，但却很少与布尔操作数一起使用。

### 布尔 XOR (^)

当与布尔操作数一起使用时，此运算符会对它的操作数执行 Exclusive OR (XOR) 操作。如果两个操作数中只有一个操作数为 true 时，所得的值才会为 true；换句话说，如果两个操作数皆为 false 或皆为 true，则所得的值便都为 false。与 &&、|| 不同的是，此运算符必须将两个操作数的值都求算出来。^ 运算符也常常被用作 bitwise 运算符，并与整数操作数一起使用。当与布尔操作数一起使用时，此运算符与 != 运算符相同。

## Bitwise 运算符与移位运算符

Bitwise 运算符与移位运算符是较低级的运算符，它们是用来对构成整数值的单独位进行操作。Bitwise 运算符通常都是用来测试与设置整数值中的单独标记位。为了要了解它们的行为，你必须了解二进制数与用来表示负数的二进制补码的格式 (twos-complement format)。你不可以将这两种类型的运算符与浮点数、布尔值、数组、对象操作数一起使用。若使用布尔操作数时，&、| 与 ^ 运算符会执行不同的操作，我们在之前就已经介绍过了。

如果 bitwise 运算符的参数中有任何一个为 long 时，则所得的结果便会为 long；否则，所得的结果就会为 int。如果移位运算符的左侧操作数为 long，则所得的结果会是 long；否则，所得的结果会为 int。这两种运算符包括了：

### bitwise 补码 (~)

一元运算符 ~ 被称作 bitwise 补码 (complement) 或 bitwise NOT 运算符。它会将操作数的每一位反相 (invert)，即把 1 转换为 0，把 0 转换为 1。例如：

```
byte b = ~12;           // ~00001100 = => 11110011 或十进制的 -13
flags = flags & ~f;     // 将标记集 flags 中的 f 标记清零
```

### bitwise AND (&)

此运算符会对它的两个整数操作数中的每一个位进行布尔 AND 操作。只有在所对应的两个位都是 1 的情况下，在所得结果中的该位才会为 1。例如：

```
10 & 7           // 00001010 & 00000111 = => 00000010 或 2
if ((flags & f) != 0) // 测试标记 f 是否已被设定
```

当与布尔操作数一起使用时，& 就成为之前所说的不经常使用的布尔 AND 运算符。

### bitwise OR (|)

此运算符会对它的两个整数操作数中的每一个位进行布尔 OR 操作。在所对应的两个位中有任何一个为 1 的情况下，所得的结果中的该位便为 1；也只有在所对应的两个位都为 0 的情况下，所得的结果中的该位才会为 0。例如：

```
10 | 7           // 00001010 | 00000111 = => 00001111 或 15
flags = flags | f; // 设定标记 f
```

当与布尔运算符一起使用时，| 就成为之前所说的不经常使用的布尔 OR 运算符。

### bitwise XOR (^)

此运算符会对它的两个整数操作数中的每一个位进行布尔 XOR (Exclusive OR) 操作。在所对应的两个位都不相同时，所得的结果中的该位才会为 1；如果相对应的两个位都是 1 或都是 0，所得的结果中的该位就是 0。例如：

```
10 ^ 7           // 00001010 ^ 00000111 = => 00001101 或 13
```

当与布尔运算符一起使用时，^ 就成为我们在前面描述的不经常使用的布尔 XOR 运算符来使用。

### 左移 (<<)

<< 运算符会将左侧操作数向左移动右侧操作数所指定的位数，因此，左侧操作数中的高位便会被丢掉，而低位则会由右侧移进左操作数中。将整数向左移  $n$  位，相当于将该数乘以  $2^n$ 。例如：

```
10 << 1          // 00001010 << 1 = 00010100 = 20 = 10*2
7 << 3           // 00000111 << 3 = 00111000 = 56 = 7*8
-1 << 2          // 0xFFFFFFFF << 2 = 0xFFFFFFFFC = -4 = -1*4
```

如果左操作数为 long，则右操作数的值必须介于 0 至 63 之间；如果为 int，则右操作数的值便必须介于 0 与 31 之间。

### 有符号的右移 (>>)

>> 运算符会将左侧操作数向右移动右侧操作数所指定的位数，左侧操作数的低位会被丢掉，最高位会被补上一个原先的左侧操作数中最高位的值。如果左操作数为正值时，零会由左移进左操作数；如果为负值，则 1 会由左移进左操作数。这种技术称作 *sign extension*，它是用来保留左操作数的符号，例如：

```
10 >> 1      // 00001010 >> 1 = 00000101 = 5 = 10/2
27 >> 3      // 00011011 >> 3 = 00000011 = 3 = 27/8
-50 >> 2     // 11001110 >> 2 = 11110011 = -13 != -50/4
```

如果左操作数为正值且右操作数的值为  $n$  时，则  $>>$  运算符的运算结果相当于将左操作数除以  $2^n$ 。

无符号的右移 ( $>>>$ )

此运算符类似于  $>>$  运算符，它只在左操作数的最高位处放置零，而不管左操作数原本的值为正或负。这种技术称为 *zero extension*。此运算符在左操作数被当作无符号数时才适合被使用（尽管 Java 的所有整数类型都为有符号数）。例如：

```
0xff >>> 4    // 11111111 >>> 4 = 00001111 = 15 = 255/16
-50 >>> 2     // 0xFFFFFCE >>> 2 = 0x3FFFFFF3 = 1073741811
```

## 赋值运算符

赋值运算符会将数值存储或赋予某个变量。左侧的操作数必须为一个适当的本地变量、数组元素或对象字段，右侧的操作数则必须是与变量兼容的数值。赋值表达式会计算出将被赋给该变量的值，但更重要的是，此表达式在执行赋值操作时会产生副作用。与其他二元运算符不同的是，赋值运算符为右结合的 (*right-associative*)，如以下的表达式： $a=b=c$  是按由右而左的方向来执行的，正如  $a=(b=c)$ 。

基本的赋值运算符为  $=$ ，请不要将它与等号运算符  $==$  混淆了。为了将它们加以区别，我建议你可以将  $=$  读成“被赋值为”。

除了这种简单的赋值运算符外，Java 还定义了其他 11 个运算符，它们是由 5 个数学运算符与 6 个 bitwise 及移位运算符所组成的。例如： $+=$  运算符会读取左侧变量的数值，并将该值与右侧的操作数相加，然后将总和存储于左边的变量中，并将该总和作为表达式的值返回。因此，表达式  $x+=2$  几乎就等于  $x=x+2$ ，但这两个表达式不同之处是当你使用  $+=$  运算符时，左侧的操作数只会被求值一次。如果该操作数具有副作用时就会有不同的结果，请看以下的两个表达式：

```
a[i++] += 2;
a[i++] = a[i++] + 2;
```

赋值运算符的一般格式为：

```
var op= value
```

此一般格式又等同于（除非  $var$  中有副作用）：

```
var = var op value
```



最常使用的运算符是 `+=` 与 `-=`，而在处理布尔标记时较常使用到 `&=` 与 `|=`。例如：

```
i += 2;           // 对循环计数器递增 2
c -= 5;           // 对循环计数器递减 5
flags |= f;       // 在一组整数 flag 中设置标记 f
flags & ~f;        // 在一组整数 flag 中清除标记 f
```

## 条件运算符

条件运算符`?:`是一个继承自C语言的三元运算符，它让你可以在表达式中嵌入条件。你可以将它想成是 `if/else` 语句的表达式形式，它的第一个与第二个操作数是用`?`所隔开，而第二个与第三个操作数是用`:`区分开。第一个操作数必须为布尔值，第二个与第三个运算符就可以是任何类型的值，但两者必须为同一个类型。

条件运算符首先会先计算第一个操作数的值，如果值为 `true`，则会接着计算第二个操作数的值，并将该值当作该表达式的值；如果第一个操作数的值为 `false`，则会计算出第三个操作数的值，并将该值当作该表达式的值。条件运算符永远不会同时计算第二个与第三个操作数的值，所以在使用具有副作用的表达式时要特别小心。此运算符的使用范例如下：

```
int max = (x > y) ? x : y;
String name = (name != null) ? name : "unknown";
```

请注意，在所有的运算符中，`?:`运算符的优先级只比赋值运算符高而已，因此在一般的情况下，并不需要为它的操作数加小括号。很多程序员发现，条件运算符中的第一个运算符若加上小括号会使阅读更容易些，就如同 `if` 语句总是将它的条件表达式加上小括号一样。

## instanceof 运算符

`instanceof` 运算符需要对象或数组值作为它的左操作数，并要求引用类型数据的名称作为它的右操作数。如果对象或数组是指定类型的实例 (*instance*)，则会返回 `true`，否则返回 `false`。如果左操作数为 `null`，则 `instanceof` 一定会返回 `false`；如果 `instanceof` 表达式所求得的值为 `true`，就表示你可以安全地强制转换并将右操作数的变量类型赋值给左操作数。

`instanceof` 运算符只可与引用类型和对象一起使用，而不能与基本数据类型一起使用。`instanceof` 的范例如下：

```
"string" instanceof String // True: 所有的字符串都是 String 的实例
"" instanceof Object        // True: 字符串同时也是 Object 的实例
null instanceof String      // False: null 不是任何东西的实例
```

```
Object o = new int[ ] {1,2,3};
o instanceof int[ ]      // True: 数组值是一个 int 数组
o instanceof byte[ ]     // False: 数组值不是一个字节数组
o instanceof Object      // True: 所有的数组都是 Object 的实例

// 使用 instanceof 来确保强制转换某个对象是安全的
if (object instanceof Point) {
    Point p = (Point) object;
}
```

## 特殊运算符

Java 共有五个语言结构被当作运算符，而有时也被当作基本语法的一部分。这些“运算符”也被列在表 2-4 中，以显示它们与其他运算符间的优先级关系。这些语言结构在本章稍后会有详细的说明，这里只简短地介绍一下，旨在在往后的程序代码中见到这样的结构时不再那么陌生。

### 对象成员访问 (.)

对象 (object) 是由一组数据与操作这些数据的方法所组成的。对象的数据字段与 method 就称为它们的成员 (member)。点号 (.) 运算符可用来访问这些成员，如果 *o* 是一个表达式，*f* 是对象的字段名称，则 *o.f* 代表该字段的值。如果 *m* 是 method 的名称，*o.m* 则是指该方法，并可使用 () 运算符加以调用。

### 数组元素访问 ([])

数组是由一连串编号的值所组成的。数组中的每个元素都可以通过其编号 (或称作索引 (index)) 来访问。[] 运算符可以让你访问到数组中的某个元素。如果 *a* 是一个数组，*i* 是计算一个 int 类型的表达式，则 *a[i]* 是指 *a* 数组中的某个元素。与其他运算符不同的是，数组索引值只能是 int 类型或位更少的类型。

### 方法调用 (())

method 是由一些 Java 程序代码所组成的。当被运行或调用时，必须在它的名称后面加上一对小括号，小括号内则可以有由零个或多个逗号所隔开的表达式，这些表达式的值就是 method 的自变量 (argument)。Method 会处理这些自变量，并会视需要而将结果返回，而所返回的值便是该方法调用表达式的值。如果 *o.m* 是一个没有自变量的 method，则该方法可以使用 *o.m()* 加以调用。例如，如果 method 需要用到 3 个自变量，就可以用像是 *o.m(x,y,z)* 这样的表达式加以调用。在 Java 解释器调用 method 之前，它会先计算出每个要传递给此 method 的自变量值，且计算的顺序是由左而右 (如果这些自变量具有副作用时，就会有不同的计算顺序，而所得的结果就会有所不同)。

### 对象的创建 (new)

在 Java 中，对象 (及数组) 的创建是使用 new 运算符，也就是在它之后接着被创

建的对象类型以及由小括号括起来需要传递给对象构造函数 (constructor) 的自变量列表。构造函数是一个特殊的 method，它会初始化一个新创建的对象，所以对象创建的语法与 Java method 调用的语法是很相似的。例如：

```
new ArrayList();
new Point(1,2)
```

### 类型转换或强制转换 (())

正如我们之前所看过的，小括号可被用于类型转换或强制转换。此运算符的第一个操作数是你想要转换成的类型，它被放在小括号内；而第二个操作数则是要被转换的值，它紧接在小括号的后面。例如：

```
(byte) 28           // 一个整数直接量转换为 byte 类型
(int) (x + 3.14f)    // 一个浮点值转换为整数值
(String)h.get(k)     // 一个一般的对象转换为一个较为特别的字符串类型
```

## 语句

语句是可以被 Java 解释器执行的单独命令。在默认情况下，Java 解释器逐一执行语句，也就是依照语句在程序代码中的顺序加以执行。但是，Java 所定义的语句中的很多都是流程控制语句，例如条件与循环，它们改变了原本默认的执行顺序。表 2-5 归纳了 Java 所定义的语句。

表 2-5: Java 语句

语句	目的	语法
expression	副作用	var = expr; expr++; method(); new Type();
compound	语句组	{ statements }
empty	不做任何事	;
labeled	命名一条语句	label : statement
variable	声明一个变量	[final] type name [= value] [, name [= value]] ...;
if	条件	if ( expr ) statement [ else statement]
switch	条件	switch ( expr ) { [ case expr : statements ] ... [ default: statements ] }
while	循环	while ( expr ) statement
do	循环	do statement while ( expr );
for	简单循环	for ( init ; test ; increment ) statement
for/in	集合迭代	for ( variable : iterable ) statement 在 Java 5.0 之后，也称作 “foreach”



表 2-5: Java 语句 (续)

语句	目的	语法
break	退出该语句块	break [ label ] ;
continue	重新开始循环	continue [ label ] ;
return	结束 method	return [ expr ] ;
synchronized	临界区	synchronized ( expr ) { statements }
throw	抛出异常	throw expr ;
try	处理异常	try { statements } [ catch ( type name ) { statements } ] ... [ finally { statements } ]
assert	验证错误	assert invariant [ : error ] ; Java 1.4 之后才有

## 表达式语句

在本章的前半段我们已经看到, Java的表达式中有些具有副作用。换句话说, 它们并不是简单地计算出某个值而已, 而且会改变程序的状态。任何带有副作用的表达式都可以被当作一个语句加以使用, 但最后要记得加上一个分号。合法的表达式语句包括: 赋值、递增或递减、方法调用与对象创建。例如:

```

a = 1;           // 赋值
x *= 2;          // 赋值与运算
i++;             // 后递增
--c;             // 前递减
System.out.println("statement"); // 方法调用

```

## 复合语句

复合语句是将一些相关的语句用一对大括号将它们括起来, 你可以在Java语法中规定需要出现语句的地方使用复合语句:

```

for(int i = 0; i < 10; i++) {
    a[i]++;           // 此循环的主体是一个复合语句
    b[i]--;           // 在一对大括号中包含了两个表达式语句
}

```

## 空语句

Java中的空语句 (empty statement) 的写法为一个单独的分号。空语句并不会做任何的事, 但这种语法有时候是非常有用的。例如, 你可以使用空语句作为 for 循环的主体:

```
for(int i = 0; i < 10; a[i++]++) // 递增数组元素
    /* empty */;                // 循环主体是一个空语句
```

## 标签语句

标签语句 (labeled statement) 是在某个语句前面加上个标识符以及一个冒号。标签在 `break` 与 `continue` 语句中会被使用到, 例如:

```
rowLoop: for(int r = 0; r < rows.length; r++) { // 一个标签循环
    colLoop: for(int c = 0; c < columns.length; c++) { // 另一个标签循环
        break rowLoop; // 使用标签
    }
}
```

## 局部变量声明语句

局部变量常常都被简称为变量, 是用来代表某个值所存放位置的符号名称, 它可以在 `method` 或复合语句中被定义。所有的变量在被使用之前都必须先声明, 而变量声明语句就是在做这件事情。因为 Java 是一种对类型要求非常强的语言, 因此变量在声明时必须被明确地指定类型, 且只有该类型的数据才能被存储于该变量中。

在最简单的形式中, 变量声明指明了变量的类型与名称:

```
int counter;
String s;
```

变量声明也可以包含初始值的设定, 即对所声明的变量设定初始值。例如:

```
int i = 0;
String s = readLine();
int[] data = {x+1, x+2, x+3}; // 数组的初始化会在稍后的章节讨论
```

Java 编译器不允许你去使用一个尚未初始化的局部变量, 所以为了方便起见, 通常会在声明变量时也一并为其设定初始值。初始值表达式并不一定需要直接量值或是一个能被编译器计算出的常量表达式, 它可以是一个任意的复杂表达式, 而其值可在程序运行中被计算出来。

一个变量声明语句可以声明或初始化一个以上的变量, 但所有的变量必须要是同一种数据类型。变量名称与非必要的初始值都必须使用逗号以跟其他的变量做区分:

```
int i, j, k;
float x = 1.0, y = 1.0;
String question = "Really Quit?", response;
```

在 Java 1.1 及之后的版本中, 变量声明语句的开头可以使用 `final` 关键字。此修饰符表示当此变量一旦设定好初始值后, 其值就不允许被任意地改变:

```
final String greeting = getLocalLanguageGreeting();
```

C语言程序员必须特别注意,Java的变量声明语句可以在Java程序代码中的任何地方出现,它并没有被限制为只能放在method或代码块的开头处。局部变量声明也可以和for循环中的初始值设定部分结合,稍后我们将会加以介绍。

局部变量只可以在该方法或代码块中被使用,我们称它为变量作用域(variable scope,或称作lexical scope):

```
void method() {                // 一个method
    int i = 0;                  // 声明变量 i
    while (i < 10) {            // 此处的 i 是在其作用域之中
        int j = 0;              // 声明 j, j 的作用域从这里开始
        i++;                    // 此处的 i 是位于作用域之中并递增 i
    }                          // j 不再位于作用域中,且不能再被使用了
    System.out.println(i);      // i 仍然位于作用域中
}                               // i 的作用域在这儿结束
```

## if/else 语句

if 语句是最基本的控制语句,它允许 Java 借此在运行程序中作出判断,或更为精确地说,是有条件地执行语句。If 语句必须有一个关联的表达式与语句,如果表达式求得的结果为 true,解释器便会执行该语句;如果表达式求得值为 false,则解释器便会忽略该语句而不再去执行它。在 Java 5.0 中,表达式可以是 wrapper 类型 Boolean 来代替基本数据类型 boolean。以下的范例就说明了 wrapper 对象是自动地 unboxed。

这是 if 语句的范例:

```
if (username == null)           // 如果 username 为 null,
    username = "John Doe";      // 便定义它的值
```

虽然它们看起来没什么关联,但由小括号所括起的表达式在 if 语句中是必要的一部分。

正如我先前所说的,可由一对大括号将其所有的语句段给括起来,所以我们可以将 if 语句写成如下的样子:

```
if ((address == null) || (address.equals("")) ) {
    address = "[undefined]";
    System.out.println("WARNING: no address specified.");
}
```

if 语句可以视情况搭配 else 关键字一起使用,而跟随在 else 关键字之后的则是第二个语句。如果 if 表达式的值为 true,则第一个语句就会被执行,否则就执行第二个语句。例如:



```
if (username != null)
    System.out.println("Hello " + username);
else {
    username = askQuestion("What is your name?");
    System.out.println("Hello " + username + ". Welcome!");
}
```

当你在使用嵌套 if/else 语句时，必须要确认与 else 同一组的 if 是哪一个。请看以下的范例：

```
if (i == j)
    if (j == k)
        System.out.println("i equals k");
else
    System.out.println("i doesn't equal j");    // 错了 !!
```

这个例子中，内部的 if 语句形成了单独的语句，这种形式是外部的 if 语句所允许的。不幸的是，和 if 同一组的 else 并没有办法很清楚地被辨识出来（除了缩排方式给出的提示之外）而且在这个例子中，缩排的提示也错了。这里就应该遵循的规则是：else 是与离它最近的 if 语句为一组，所以将上述的程序代码缩排方式稍加调整：

```
if (i == j)
    if (j == k)
        System.out.println("i equals k");
else
    System.out.println("i doesn't equal j");    // 错了 !!
```

这是合法的程序代码，但却不能很明确地表达出程序员所考虑的问题。当你在使用嵌套的 if 语句时，必须使用大括号来确保你的程序代码容易被阅读。以下是较好的写法：

```
if (i == j) {
    if (j == k)
        System.out.println("i equals k");
}
else {
    System.out.println("i doesn't equal j");
}
```

## else if 子句

if/else 语句相当有用，可以用来测试条件的成立与否，并在两个语句或代码块间选择其一来加以执行。但如果所需选择的语句或代码块不只两个的时候你该怎么办呢？一般的做法是使用 else if 子句。它的写法如下：

```
if (n == 1) {
    // 执行代码块 #1
}
else if (n == 2) {
```

```
        // 执行代码块 #2
    }
    else if (n == 3) {
        // 执行代码块 #3
    }
    else {
        // 如果所有的 else 皆为 false, 则执行代码块 #4
    }
}
```

上面的程序代码并没有什么特殊之处,它是由一连串的 if 语句所构成的。从第二个 if 开始,每一个 if 语句都是前一个语句的 else 子句。使用 else if 的写法更好些,同时也比以下的程序代码更具可读性:

```
if (n == 1) {
    // 执行代码块 #1
}
else {
    if (n == 2) {
        // 执行代码块 #2
    }
    else {
        if (n == 3) {
            // 执行代码块 #3
        }
        else {
            // 如果所有的 else 皆为 false, 则执行代码块 #4
        }
    }
}
```

## switch 语句

switch 语句会在程序的执行流程中产生一个分支 (branch)。当然,你可以使用前一小节介绍的多重 if 语句来实现多路分支,但这并不是最好的解决方法,特别是当所有的分支都决定于同一个变量的值时。在这种情况下,如果使用多重 if 语句来对同一个变量做重复地测试是很没有效率的。

有一个较好的解决方法就是使用 switch 语句,它继承自 C 语言。虽然该语句的语法不像 Java 的其他语句那样优雅,但在实际上却具有相当的实用性。或许你对 switch 语句还不熟悉,但可能已经不知不觉对它的基本概念有些许的了解了。

switch 语句由一个表达式开始,它可以是 int、short、char 或 byte 类型的。在 Java 5.0 中,它也可以使用 Integer、Short、Character 与 Byte wrapper 类型,枚举类型 (Enumerated type, Enums 是 Java 5.0 中新加入的一个类型,在第四章将会对枚举类型及其在 switch 语句里的用法做更详细的介绍) 也一样。紧接在表达式后面的是由一对大括号括起来的代码块,它包含了许多进入点 (entry point),且该进入点有可能

和表达式经过计算后的值相对应。例如，以下的 switch 语句相当于前一小节所介绍的重复 if 与 else/if 语句的程序代码：

```
switch(n) {
    case 1:                // 如果 n == 1, 从这儿开始
        // 执行代码块 #1
        break;            // 在这儿停止
    case 2:                // 如果 n == 2, 从这儿开始
        // 执行代码块 #2
        break;            // 在这儿停止
    case 3:                // 如果 n == 3, 从这儿开始
        // 执行代码块 #3
        break;            // 在这儿停止
    default:               // 如果所有的 case 皆不符合条件……
        // 执行代码块 #4
        break;            // 在这儿停止
}
```

如同你在例子中所看到的，switch 主体的所有进入点都是以关键字 case 紧跟着整数与一个冒号，或是一个特殊 default 关键字跟着一个冒号来区分的。当 switch 语句被执行时，解释器会去计算出小括号内表达式的值，然后在所有 case 标签中寻找匹配的值，如果找到了，解释器会开始执行 case 标签后的代码块；如果在 case 标签里没有找到一个匹配的值时，解释器就会执行 default: 后的第一个语句；如果没有 default: 标签，则解释器会忽略整个 switch 语句的主体。

请注意在程序代码中位于每一个 case 结尾的 break 关键字，break 会使得解释器退出 switch 语句主体。在 switch 语句中的 case 子句只会指出将会被执行的程序代码的起始点 (starting point)，每一个 case 并不是独立的代码块，而且它们没有明确的结束点 (ending point)。因此，你必须使用 break 或相关的语句来明确指出每一个 case 的结尾。如果没有 break 语句，switch 语句会从与表达式求出的值相同的 case 标签开始执行，直到 switch 主体结束为止。在极少数的情况之下，才会用这样的方式编写程序，但在 99% 的状况中你必须在每一个 case 或 default 区域的最后加上一个可以让 switch 语句结束执行的语句。一般都会使用 break 语句，但 return 与 throw 也具有相同的功能。

switch 语句可以带一个以上的 case 子句，请看以下 method 中的 switch 语句：

```
boolean parseYesOrNoResponse(char response) {
    switch(response) {
        case 'y':
        case 'Y': return true;
        case 'n':
        case 'N': return false;
        default: throw new IllegalArgumentException("Response must be Y or N");
    }
}
```



switch 语句和它的 case 标签有一些要注意的地方。首先，与 switch 语句相关的表达式必须有一个 byte、char、short 或 int 类型的值，浮点与布尔类型的是不能使用的，而 long 虽然是整数类型，但也能使用。第二，每一个 case 标签后面的值必须为常量或可被编译器计算出值的常量表达式，也就是不能含有变量或 method 调用。第三，case 标签值必须为 switch 表达式所使用的数据类型能表示的范围。最后，有两个或两个以上的 case 标签值拥有相同的值，或有超过一个的 default 标签都是不合法的。

## while 语句

它和 if 语句一样是 Java 基本的流程控制语句，是一个可让 Java 重复地执行某个动作的基本语句。它的语法如下：

```
while (expression)
    statement
```

while 语句在执行时会先计算出表达式的值，它必须是布尔值（或者是在 Java 5.0 中的 Boolean）。如果计算出来的值为 false，解释器会忽略掉该循环内的语句并跳到下一个语句；如果计算出来的值为 true，则循环主体内的语句就会被执行，且该表达式会再次被计算。再一次地，如果计算出的值为 false，则解释器会跳到下一个语句去执行；否则的话，就会再次执行循环主体内的语句。当 while 表达式求得的值为 true，循环就会不断地被执行（直到值为 false 才会停止）。当 while 语句结束后，解释器会跳至下一个语句。你可以使用 while(true) 创建一个无限循环。

这是一个会显示 0 到 9 的 while 循环范例：

```
int count = 0;
while (count < 10) {
    System.out.println(count);
    count++;
}
```

正如你所看到的，变量 count 由 0 开始，且在每次循环主体被执行时就会进行递加操作。当循环执行了 10 次，表达式就会变成 false (count 不再小于 10)，同时 while 语句执行完毕且 Java 解释器会移到下一个语句。大部分的循环都会有一个像 count 一样的计数器变量，虽然变量 i、j 与 k 是循环计数器常使用的变量，但我们还是建议使用较具意义的名称，好让程序更容易被了解。

## do 语句

do 循环和 while 循环非常的相似，但是它的循环表达式在循环的底部才会被测试，而不是在顶部，也就是说循环的主体至少会被执行一次。它的语法如下：

```
do
    statement
while ( expression ) ;
```

do 循环与 while 循环有几个不同点是要特别注意的。首先，do 循环必须使用 do 关键字作为循环的开始，并且使用 while 关键字作为循环的结束并引入循环的条件。与 while 循环不同的是，do 循环的结束必须有一个分号，这是因为 do 循环是以循环的条件作为结束，而不是单单使用一个大括号来表示循环主体的结束。以下是使用 do 循环显示出与上述 while 循环相同的结果：

```
int count = 0;
do {
    System.out.println(count);
    count++;
} while(count < 10);
```

do 循环的使用机会比 while 循环来得低，因为实际上，你很少会遇到至少需执行一次循环主体的情况。

## for 语句

for 语句提供了一个比 while 与 do 循环更为方便的循环结构，它利用了一般循环的执行模式。大部分的循环都有一个计数器或某些种类的状态变量，在循环开始之前它们就要被初始化，然后加以测试并决定是否要执行循环的主体，且在下次表达式被计算前必须递增或更新其值。初始化、测试以及更新，这三个步骤是循环变量的重要操作，for 语句让这三个步骤成为循环语法中相当明确的组成部分：

```
for(initialize ; test ; update)
    statement
```

此 for 循环基本上和以下的 while 循环相同（注2）：

```
initialize;
while(test) {
    statement;
    update;
}
```

将 *initialize*、*test* 与 *update* 表达式放在 for 循环的开始位置，会让人更容易了解循环在做什么，同时它会防止错误情况的发生，如忘了将循环变量初始化或更新。解释器会忽略 *initialize* 与 *update* 表达式计算出来的值，所以 *initialize* 与 *update* 表达式必须具有副作用。通常，*initialize* 是一个赋值表达式，而 *update* 是一个递增、递减或其他类型的赋值表达式。

注2： 当我们稍后讨论到 continue 语句时你将会看到，该 while 循环并不完全等于 for 循环。

以下的 for 循环会显示出数字 0 到 9，与先前的 while 与 do 循环的例子一样：

```
int count;
for(count = 0 ; count < 10 ; count++)
    System.out.println(count);
```

此语句的语法将循环变量中所有重要的数据都放在同一行内，让人相当容易了解循环的执行方式。将 *update* 表达式放在 for 语句里，同样也简化了循环的主体，甚至我们都不须使用大括号。

for 循环也支持一些语法让它更易于使用，因为许多循环只在循环里使用循环变量，因此 for 循环允许 *initialize* 表达式是一个完整的变量声明，而该变量的作用范围便仅限于循环之内。例如：

```
for(int count = 0 ; count < 10 ; count++)
    System.out.println(count);
```

此外，for 循环语法并没有限制在循环内只能使用一个变量，for 循环可以使用逗号将多个 *initialization* 与 *update* 表达式分隔开。例如：

```
for(int i = 0, j = 10 ; i < 10 ; i++, j--)
    sum += i * j;
```

虽然所有的例子都是使用数字来计数，但 for 循环并没有限制必须使用数字来控制循环。例如，你可以通过迭代（iterate）一个链接列表（linked list）中的元素来计数：

```
for(Node n = listHead; n != null; n = n.nextNode())
    process(n);
```

*initialize*、*test* 与 *update* 表达式都不是一定要有的，只有用来分隔表达式的分号是必须的。如果 *test* 表达式被省略了，则其默认值为 true，因此你可以使用 for(;;) 来编写无限循环。

## for/in 语句

for/in 语句是 Java 5.0 中新加入的具有强大的循环功能的语句。它通过数组的元素、集合、可以实现 java.lang.Iterable 的对象（我们立马就会看到很多关于这个新的接口）来不断地重复执行。每一个 iteration 会赋予一个数组元素或 Iterable 对象给你所声明的循环变量，然后执行该循环主体，该循环主体一般都是使用循环变量来操作里面的元素。for/in 语句中不需要使用到任何的循环计数器或 Iterable 对象，它会自动地重复执行，并且你不需要为初始值正确与否或循环结束而担心。

for/in 循环后面紧接着的是一个左小括号、变量的声明（不需做初始化）、冒号、表达式、一个右小括号，最后一个就是它的语句（或代码块）。这就是此循环主体的格式。



```
for( declaration : expression )  
    statement
```

for/in 循环并没有使用关键字 in，且一般都会将该循环内的冒号读成“in”。因为这个语句没有属于自己的关键字，因此就不会和另一个 for 循环造成混淆。或许你也会看到有人称它为“enhanced for”或“foreach”。

我们已经使用了 while、do 与 for 循环来编写可以显示出 10 个数字的范例，同样地，for/in 循环也可以做到，但不能仅靠该循环本身。for/in 是不同于其他一般用途的一个很特殊的循环；它会对每一个数组或集合内的元素执行一次该循环主体，所以如果要执行循环 10 次（显示出 10 个数字），我们就需要拥有 10 个元素的数组或集合。

以下是可供我们参考的程序代码：

```
// 这些是我们想要显示出来的数字  
int[] primes = new int[] { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };  
// 这是可以将它们显示出来的循环  
for(int n : primes)  
    System.out.println(n);
```

以下是关于 for/in 循环，你所必须知道的语法：

- *expression* 必须是数组或可以实现 java.lang.Iterable 接口的对象。这个类型必须在编译时就被确定了，编译器可以产生合适的循环程序代码。例如，不可以在循环中使用经由强制转换成 Object 类型的数组或 List。
- 数组类型或 Iterable 元素必须要和 declaration 中声明的变量类型兼容。如果你使用了一个 Iterable 对象，且它的元素类型没有被参数化时，则该变量就必须被声明为 Object（parameterized 类型也是 Java 5.0 的新功能，在第四章会有详细的说明）。
- *declaration* 通常由一个类型和一个变量名称组成，但它也可以加入一个 final 修饰符及一些适当的注释（请看第四章）。使用 final 可以避免循环变量使用由循环所赋予的数组或集合元素之外的其他值，并帮助强调循环变量中的数组或集合不可被更改。
- for/in 循环的循环变量的类型与变量的名称必须被声明为循环的一部分，同时不能像 for 循环一样将变量的声明放在循环的外面。

在后面的章节会更进一步地说明 for/in 语句的用法。它和 parameterized 类型之间的关系在第四章都会介绍。

```
import java.util.*;  
public class ForInDemo {
```

```
public static void main(String[] args) {
    // 这是一个 collection, 我们将会不断地重复以下的操作
    Set<String> wordset = new HashSet<String>();

    // 一个基本循环从这里开始, 不断地重复着数组的元素
    // 循环主体会对每一个 args[] 内的元素执行一次操作
    // 每一次会有一个元素被赋给变量 word
    for(String word : args) {
        System.out.print(word + " ");
        wordset.add(word);
    }
    System.out.println();

    // 现在逐一处理 Set 的元素
    for(String word : wordset) System.out.print(word + " ");
}
```

## Iterable 与 iterator

为了了解 `for/in` 循环如何与 `collection` 一起运作, 我们必须看一下这两个接口 (interface), `java.lang.Iterable` (Java 5.0 中新加入的) 与 `java.util.Iterator` (Java 1.2 中新加入的), 而 `Collection Framework` 其他部分的参数化是在 Java 5.0 中才有的 (注3)。为了方便起见, 我们在这里将这两个接口的 API 显示出来:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

`Iterator` 定义了一种方式来迭代集合的元素或其他的数据结构。它的运作方式是这样的: 当有很多元素在 `collection` (`hasNext()` 返回值为 `true`) 中时, 你可以调用 `next()` 来取得下一个 `collection` 中的元素。有序的 `collection` 就和 `list` 一样, 一般 `iterator` 会保证它们返回的值也是有序的。`Set` 就是无序的 `collection`, 它只会保证重复的调用 `next()` 将 `set` 内所有的元素返回来而不会有所遗漏或重复, 但不具有顺序性。

```
public interface Iterable<E> {
    java.util.Iterator<E> iterator();
}
```

使用 `Iterable` 接口来让 `for/in` 循环运行。使用类来实现这个接口, 是为了向任何一个有兴趣的对象告知它们提供了 `Iterator` (这在它们自己的权限内, 是可以有效地被

---

注3: 如果你对于 `parameterized` 类型并不熟悉的话, 可能要先跳过这一小节, 在读完第四章后再回过头来继续了解它。

使用的,即使你没有使用 for/in 循环)。如果有一个对象为 `Iterable<E>`,就表示它有一个 `iterator()` method,并会返回 `Iterator<E>`,同时它也会有一个 `next()` method,并会返回类型为 `E` 的对象。如果你实现了 `Iterable` 并为你的类提供了 `Iterator`,就可以使用 for/in 循环来迭代处理这些类。

如果你将 for/in 循环与 `Iterable<E>` 一起使用,则此循环变量必须为类型 `E` 或它的父类或接口。例如,通过迭代 `List<String>` 元素,则该变量必须被声明为 `String` 或它的父类 `Object`,或是它的接口 `CharSequence`、`Comparable` 以及 `Serializable` 中的一个。

如果你使用了 for/in 循环并迭代没有类型参数的 `List` 元素,则 `Iterable` 与 `Iterator` 也不具有类型参数,且该类型是通过 `Object` 类型的 `Iterator` 的 `next()` method 返回。在这样的状况下,你只能将循环变量声明为 `Object`。

### for/in 无法做到的事

for/in 是一个很特殊的循环,它可以简化你的程序代码,同时许多情况下,它可以减少循环所可能产生的错误。但它并不能完全取代 while、for 与 do 循环,因为它将循环计数器或 `Iterator` 给隐藏了起来。也就是说,有些算法是无法利用 for/in 循环来表示的。

假设你想要列出数组中的元素并用逗号将它们区分开来。你必须在列出数组中的每一个元素之后列出逗号,除了最后一个元素不需这样做;同样地,你也可以在列出每一个元素之前将逗号给列出来,除了第一个元素之外。传统的 for 循环的程序代码会像这样:

```
for(int i = 0; i < words.length; i++) {
    if (i > 0) System.out.print(", ");
    System.out.print(words[i]);
}
```

这是一个非常简单的程序,但你却无法使用 for/in 来完成它。问题就在于 for/in 循环无法提供一个循环计数器或任何其他的方法来告知你是否处于第一个、最后一个或任何一个迭代操作中。以下是两个简单的循环,它们都无法转换为 for/in 循环,就是因为这样的原因:

```
String[] args; // 在其他的地方初始化
for(int i = 0; i < args.length; i++)
    System.out.println(i + ": " + args[i]);

// 将 words 对应至它们出现的位置
List<String> words; // 在其他的地方初始化
Map<String,Integer> map = new HashMap<String,Integer>();
for(int i = 0, n = words.size(); i < n; i++) map.put(words.get(i), i);
```



当使用 `for/in` 循环来迭代集合的元素时，也存在着相同的问题。`for/in` 循环在运用数组时没有办法获得当前数组元素的索引值；`for/in` 循环运用在 `collection` 上时也会没有办法获得 `Iterator` 对象，该对象是用来详细列举 `collection` 中的元素。也就是说，如果你使用了 `Iterator`，那么你将无法使用 `iterator` 的 `remove()` method。

还有一些是 `for/in` 循环无法做到的：

- 逆向迭代数组或 `List` 的元素。
- 使用一个循环计数器来访问不同的两个数组中具相同索引值的元素。
- 通过 `List` 的元素来调用 `get()` method，而不是调用 `iterator` 来迭代。

## break 语句

`break` 语句会让 Java 解释器立即终于当前正在运行中的代码块，我们在 `switch` 语句的介绍中已经看到过 `break` 语句的用法。`break` 语句最常被使用的方式就是在 `break` 关键字的后方加一个分号：

```
break;
```

当这样使用时，它会让 Java 解释器立刻退出它所在的 `while`、`do`、`for` 或着是 `switch` 语句。例如：

```
for(int i = 0; i < data.length; i++) {    // 遍历数组
    if (data[i] == target) {              // 当我们找到我们所要值时，
        index = i;                        // 将我们找到的值记录起来
        break;                            // 并且停止搜索
    }
}                                           // Java 解释器会在停止运行后来这儿
```

`break` 语句的后面也可以接着一个条件标签的名称。当以这样的形式使用 `break` 时，它会让 Java 解释器立刻离开该标签名称所指定的代码块，而该代码块可以为任何种类的语句，并不仅仅限于循环或 `switch`。例如：

```
testformnull: if (data != null) {          // 如果数组已定义，
    for(int row = 0; row < numRows; row++) { // 就遍历一个维度，
        for(int col = 0; col < numcols; col++) { // 接着遍历另一个维度，
            if (data[row][col] == null)         // 如果数组没有数据，
                break testformnull;             // 就把数组当作未定义
        }
    }
}                                           // Java 解释器会在停止执行 testformnull 后转到这儿
```

## continue 语句

break 语句会退出循环，continue 语句则会终止此次循环的迭代，开始下一次循环的迭代。continue 语句可加上标签一起运行，而且它也可以在 while、do 或 for 循环中被使用。当 continue 没有加上标签时，它会让最内层的循环开始执行下一次迭代；如果标签所指向的是某一个循环时，则它会使得该循环开始该标签所指定的循环的下次迭代。例如：

```
for(int i = 0; i < data.length; i++) {    // 循环处理数据
    if (data[i] == -1)                    // 如果数据不存在，
        continue;                        // 则跳到下一个循环
    process(data[i]);                    // 处理该数据值
}
```

while、do 与 for 循环的 continue 运作方式有些许的不同：

- 当与 while 循环一起使用时，Java 解释器会直接回到循环的最前面，再次测试循环的条件，如果计算出来的值为 true，则会再次执行循环的主体。
- 当与 do 循环一起使用时，解释器会跳至循环的最底部，并测试循环的条件以决定是否要执行下一次循环的迭代。
- 当与 for 循环一起使用时，解释器会跳至循环的最前面，在那儿它会先计算出 update 表达式的值并计算 test 表达式，以决定是否要再次执行循环。你可以发现，for 循环搭配 continue 语句一起执行时的行为与 while 循环搭配 continue 一起执行时的行为是不一样的；在 for 循环中 update 会被求出值，但在 while 循环中则不会。

## return 语句

return 语句会告诉 Java 解释器停止执行当前的 method。如果此 method 被声明为具有返回值，则 return 语句后就必须跟着一个表达式，而此表达式计算出的结果就会是此 method 的返回值。例如，以下的 method 会计算并返回一个数字的平方：

```
double square(double x) {    // 一个用来计算 x 平方的 method
    return x * x;            // 计算并返回一个值
}
```

有些 method 被声明为 void，这表示它们没有任何的返回值。Java 解释器执行这些 method 时，会依序执行它的语句直到 method 结束。在执行了最后一条语句之后，解释器便会返回原调用程序。有时候，void method 在执行最后一个语句前就必须返回，此时可以使用 return 语句，而不须要加上任何的表达式。例如，以下的 method 便会输出值，但不会返回其自变量的平方根；如果该自变量为负数，就不会输出任何的值：

```
void printSquareRoot(double x) {           // 一个会输出x平方根的 method
    if (x < 0) return;                     // 如果x是负值, 则立刻返回
    System.out.println(Math.sqrt(x));      // 输出x的平方根
}                                           // method 的结束: 隐舍地返回
```

## synchronized 语句

使用 Java 写多线程的程序是一件很容易的事 (范例请见五章), 但当你在使用多个线程时必须时时小心防止多个线程同时去修改某个对象, 使得对象的状态发生冲突。对于不能被同时执行的代码块, 我们称它为临界区 (critical section), Java 提供了 `synchronized` 语句来保护临界区, 其语法为:

```
synchronized ( expression ) {
    statements
}
```

*expression* 所代表的是一个表达式, 它必须是对象或数组; *statement* 则是由临界区的程序代码所组成的, 且必须用大括号将它们括起来。在执行临界区之前, Java 解释器首先必须获得一个互斥锁 (exclusive lock), 该互斥锁是针对 *expression* 所指定的对象或数组, 它会一直拥有该锁直到该临界区被执行完毕为止, 然后才会将互斥锁释放。当某个线程拥有某个对象的锁时, 其他的线程便无法获得该对象的锁。因此, 没有任何的线程可以执行该临界区或其他需要该对象锁的临界区。如果线程无法立刻获得执行临界区所需的锁时, 它会一直等待, 直到该锁是可以使用的状态为止。

请注意, 一般的情况下是不需要使用 `synchronized` 语句的, 除非你的程序创建了共享数据的多线程。如果只有一个线程在访问一个数据结构的话, 就没有必要使用 `synchronized` 来保护它。当你必须使用 `synchronized` 时, 可能就会有以下的程序代码:

```
public static void SortIntArray(int[] a) {
    // 将数组 a 排序。此为 synchronized 语句,
    // 所以当我们正在排序数组时, 其他的线程无法改变数组中的元素
    // (至少没有任何的线程使用 synchronized 来保护它们对该数组所做的变动)
    synchronized (a) {
        // 在这里将数组排序
    }
}
```

在 Java 中, `synchronized` 关键字经常被拿来当作一条语句的修饰符, 当它被用在 `method` 上时, 就表示整个方法是一个临界区。对一个 `synchronized` 类方法 (class method, 即静态 method) 而言, 在执行此 `method` 之前, Java 必须先获得该类的互斥锁; 对一个 `synchronized` 实例方法 (instance method) 而言, Java 则必须先获得该类实例 (class instance) 的互斥锁 (class 与 instance method 将在第三章做详细介绍)。



## throw 语句

所谓异常 (exception) 是一个信号, 用来指出某种异常状况或错误已经发生的情况。所谓抛出 (throw) 异常, 即是发出信号以显示意外状况的发生, 而捕获 (catch) 异常则是意外状况或错误的处理——看要执行哪种操作来修复它。

在 Java 中, throw 语句是用来抛出一个异常:

```
throw expression ;
```

*expression* 所计算出的值必须是一个用来描述该异常或发生错误的异常对象 (exception object)。我们将会讨论更多种类的异常, 但现在你所必须了解的是, 异常是由对象来代表的。这里有一段程序代码, 它会抛出一个异常:

```
public static double factorial(int x) {  
    if (x < 0)  
        throw new IllegalArgumentException("x must be >= 0");  
    double fact;  
    for(fact=1.0; x > 1; fact *= x, x--)  
        /* empty */ ;           // 请注意空语句的使用  
    return fact;  
}
```

当 Java 解释器执行 throw 语句时, 它会立刻停止正常的程序运行, 并开始查找一个能够捕获或处理该异常的异常处理器 (exception handler)。异常处理器必须使用 try/catch/finally 语句, 这会在下一节加以介绍。Java 解释器首先会在该代码块中找寻异常处理器, 如果有的话, 就会马上运行异常处理器。在执行完毕后, 解释器会继续运行异常处理器代码之后的程序代码。

如果程序代码中没有适当的异常处理器, 解释器便会在更外的一层代码块中继续寻找, 如果还是没有找到, 则会继续往更外一层的程序代码去寻找, 直到找到为止。如果该 method 没有包含可以处理由 throw 语句抛出的异常的异常处理器时, 解释器会停止执行当前的 method 并回到调用该方法的程序代码处, 然后解释器会继续寻找异常处理器。Java 解释器会不断地将其堆栈中的数据取出, 以寻找异常处理。如果到最后还是无法找到, 解释器就会回到程序的 main() method, 如果依旧无法找到异常处理器时, Java 解释器就会输出错误的信息, 并将刚刚所搜寻过的堆栈踪迹显示出来, 以指出该异常发生的位置, 最后结束该程序。

## 异常类型

在 Java 里, 异常 (exception) 是一个对象, 而此对象的类型是 java.lang.Throwable,

或者更简单地说, Throwable 中的一些子类 (subclass) (注 4) 是特别用来描述所发生的异常的类型。Throwable 有两个标准的子类: java.lang.Error 与 java.lang.Exception。Error 中的子类所代表的异常, 一般来说都是无法恢复的问题, 如虚拟机用完了所有的内存或是某个类文件已被破坏而无法读取。像这样的异常都可被捕获并处理, 但一般我们并不会这样做。Exception 中的子类所代表的异常, 则是用来指出较不严重的状况。这些异常可被合理地捕获及处理, 如 java.io.EOFException, 它是已到了文件最末端的信号, 而 java.lang.ArrayIndexOutOfBoundsException, 则表示该程序试图要去读取超过数组最末端的元素。本书中, 我使用了“异常”这个词来表示任何异常对象, 不论该类型是 Exception 还是 Error。

因为每一个异常都是一个对象, 因此它可以含有数据, 且它的类可以定义 method 来操作数据。Throwable class 以及它所有的子类, 都拥有一个 String 字段来存放描述异常情形发生时的可以让人读取的错误信息。当异常对象被创建时, 异常信息也同时被设定, 并可使用 getMessage() method 加以读取。大部分的异常都只含有一个信息, 但也有少数的异常额外加入了一些数据。例如: java.io.InterruptedIOException 加入了一个名为 bytesTransferred 的字段, 用来指出有多少的输入或输出在该异常状况发生之前已被处理完毕。

## try/catch/finally 语句

try/catch/finally 语句是 Java 的异常处理机制: try 子句会建立处理异常的代码块, 此 try 代码块后面可能会跟随着零个或多个 catch 子句, 每一个子句都是用来处理某个特定类型的异常的语句; catch 子句之后则可跟随着 finally 代码块, 不管 try 代码块发生了什么状况, 该代码块包含的清理程序代码 (cleanup code), 都会被执行到。catch 与 finally 子句都不是必要的, 但每个 try 代码块都必须包含两者中的一个。try、catch 与 finally 代码块都是由大括号作为开始与结束, 且这些都是在语法中规定的, 即使该子句只包含了一条语句也不能将它们给忽略。

以下的程序代码说明了 try/catch/finally 语句的语法与用法:

```
try {  
    // 正常来说, 此程序代码会由此代码块的最顶端执行到最底端  
    // 而不会遭遇到问题。但有时候会抛出异常,  
    // 不是直接使用一个 throw 语句, 就是间接地调用  
    // 一个会抛出异常的 method  
}
```

注 4: 我们尚未讨论到子类, 这在第三章中会详加说明。

```
catch (SomeException e1) {  
    // 此代码块所包含的语句可以处理异常类型为 SomeException 或该类型的子类  
    // 此代码块中的语句可以引用名称为 e1 的异常对象  
}  
catch (AnotherException e2) {  
    // 此代码块所包含的语句可以处理异常类型为 AnotherException 或该类型的子类  
    // 此代码块中的语句可以引用名称为 e2 的异常对象  
}  
finally {  
    // 此代码块所包含的语句一定会在我们离开 try 子句之后被执行,  
    // 不管我们是如何离开的:  
    // 1) 正常的情况, 在到达该程序代码块的底端之后;  
    // 2) 因为一个 break、continue 或 return 语句;  
    // 3) 因为一个由上面的 catch 子句所处理的异常;  
    // 4) 因为一个没被处理到的异常。  
    // 但是, 如果 try 子句调用了 System.exit(),  
    // 则解释器会在 finally 子句被执行之前离开  
}
```

## try

try 子句基本上是建立一个代码块, 用来处理它自己的异常, 或是提供一些特殊的清理程序代码 (cleanup code) 来在因任何原因而终止时运行。try 子句本身并没有做什么工作, 由 catch 与 finally 子句分别做异常处理与清理的操作。

## catch

try 代码块后面可能会带有零个或多个 catch 子句, 在这里会提供处理各种不同类型的异常的程序代码。每一个 catch 子句都会与一个自变量一起被声明, 该自变量则是用来指出该子句所能够处理的异常类型, 并提供一个名称让该子句能访问到它所处理的异常对象。catch 子句所能处理的异常类型与名称和传递给 method 的自变量类型与名称必须完全相同, 除此之外, catch 子句的自变量类型必须为 Throwable 或 Throwable 的一个子类。

当一个异常被抛出时, Java 解释器会寻找一个 catch 子句, 它的自变量类型必须与该异常对象的类型相同。解释器会调用它所找到的第一个符合条件的 catch 子句, 该 catch 子句内的程序代码的动作必须能够处理该异常。例如, 如果异常是 java.io.FileNotFoundException, 则你的处理方式可能是要求用户确认输入的名称是否有误, 同时重新输入一次。我们并不需要对每个发生的异常编写 catch 子句, 因为有些时候就是要让异常向上传递给调用的 method 去捕获。例如, 像是 NullPointerException 所产生的程序错误就不应该去捕获它, 而是交由 Java 解释器列出堆栈内容与错误信息后便结束。



## finally

finally子句一般都是接在try子句之后,作为清理之用(如关闭文件、结束网络连接等)。它有用的地方在于如果try块的任何一部分被执行,且不论try块是怎么完成的,finally内部的程序代码一定会被执行。事实上,try子句要想不执行finally子句而离开的唯一方式,就是使用System.exit() method,它会让Java解释器立刻停止执行。

在一般的情况下,执行到try块的最末端后会继续执行finally块来做必要的清理工作。如果因为return、continue或break语句而离开了try块,finally块里的代码会先被执行,之后才转到新的目标代码继续执行。

如果在try块里发生了异常,而且有相关的catch块可以处理这样的异常时,控制权会先转到此catch块,然后再转到finally块。如果没有本地的catch块可以处理这样的异常时,控制权则会转到finally块,然后将异常传递给最近且可以处理此异常的catch子句。

如果finally块使用return、continue、break、throw语句或是调用一个会抛出异常的method来转移控制权时,原先未决的控制权转移就会中止,改为执行新的控制转移。例如,如果finally子句抛出一个异常,则该异常会取代任何正在被抛出的异常,如果finally子句执行了return语句,则该method会以正常的方式返回,即使有某个异常已被抛出且尚未被处理。

try与finally也可以在没有异常或catch子句的情况下一起使用,在这样的情况下,finally块就只是负责清理程序代码,不管try子句里使用的是break、continue还是return语句,它都一定会被执行到。

在之前所讨论到的for与continue语句里,我们知道for循环不能直接转换成while循环,因为continue在for循环与while循环中的运行方式有些不同。finally子句提供了一个方式让我们可以写出while循环,此while循环使用了与for循环相同的方式来处理continue语句。请看以下的for循环:

```
for( initialize ; test ; update )
    statement
```

以下的while循环的行为与上述for循环相同,即使statement块中包含了一个continue语句也一样:

```
initialize ;
while ( test ) {
    try { statement }
    finally { update ; }
}
```

然而，`finally`块中的 `update` 语句会使得此 `while` 循环碰到 `break` 语句后的动作与 `for` 循环的有所不同。

## assert 语句

`assert` 语句是用来证明与验证 Java 程序代码中的设计设想的。这个语句是 Java 1.4 中新加入的，而之前的版本都不曾使用。`assertion`是由 `assert` 关键字与其后的布尔表达式组合而成的，程序员深信该表达式的值应该都是 `true`。在默认的情况下，`assertion` 是被关起来的，这会使 `assert` 语句不会被执行到。你也可以将 `assertion` 开启，将它当成调试或测试的工具。当 `assertion` 被开启时，`assert` 语句会去计算表达式的值，如果它的值为 `true`，`assert` 就不会做任何的事情；如果该表达式的值为 `false` 时，就表示该程序已经处于不正常的状态下，这时 `assert` 语句会抛出一个 `java.lang.AssertionError`。

`assert` 语句中也可以加入第二个表达式，同时使用冒号来将它与第一个表达式分隔。当 `assertion` 在开启的状态下且第一个表达式所计算出来的结果为 `false` 时，第二个表达式的值就会被当成是错误代码或错误信息，并将这个结果作为 `AssertionError()` 构造函数的参数。完整的 `assert` 语句语法如下：

```
assert assertion ;
```

或：

```
assert assertion : errorcode ;
```

一定要记住，`assertion` 必须是一个布尔表达式，一般来说，它包括了比较运算符与使用布尔值的 `method`。

## 编译 assertion

因为 `assert` 语句是 Java 1.4 才加入的，同时也因为在 Java 1.4 之前 `assert` 并不是一个保留字，所以在这样的编译环境下使用这个新的语句，会损坏使用“`assert`”作为标识符的程序代码。基于此原因，在默认的情况下，`javac` 编译器无法识别 `assert` 语句。在编译使用 `assert` 语句的 Java 程序代码时，你必须使用命令行参数 `-source 1.4`。例如：

```
javac -source 1.4 ClassWithAssertions.java
```

Java 1.4 里，在 `-source 1.4` 被指明的情况下，`javac` 编译器才允许“`assert`”被当作是一个标识符来使用。而在一般默认的情况下，如果发现 `assert` 被当成标识符，不兼容警告信息就会产生，以促使你去修改程序代码。

Java 5.0中, `javac` 编译器在默认的情况下就可识别 `assert` 语句 (如同所有其他 Java 5.0 的新语法一样), 而且编译器在编译程序代码时不需要包含有 `assertion` 的特殊自变量。如果还有其他的程序仍然使用 `assert` 作为标识符的话, 在 Java 5.0 默认的情况下将不再编译。如果你无法修改的话, 可以在 Java 5.0 中选择性地使用 `-source 1.3` 参数。

## 启用 assertion

`assert` 语句的编码是假设应该都会为 `true`。出于性能上的考虑, 程序代码在运行时每一次都测试 `assertion` 是没有意义的。在默认的情况下, `assertion` 是被关闭的, 所以 `assert` 语句不会影响到程序。然而, `assertion` 程序代码会一直存在于 `class` 文件里, 所以当它的状态是测试、诊断与调试时, 可以一直被开启。你可以使用 Java 解释器的命令行参数来开启 `assertion`。除了系统类以外, 其他所有的类都可以使用 `-ea` 参数来开启 `assertion`, 系统类则是使用 `-esa` 参数来开启 `assertion`。对于一些特殊的类, 你可以使用 `-ea` 后面加上一个冒号及类名来开启 `assertion`:

```
java -ea:com.example.sorters.MergeSort com.example.sorters.Test
```

要为 `package` 中所有的类与它所有的 `subpackage` 开启 `assertion`, 则在 `-ea` 参数之后加上一个冒号、包名与三个点号:

```
java -ea:com.example.sorters... com.example.sorters.Test
```

你可以通过 `-da` 参数利用相同的方式来关闭 `assertsion`。例如, 为一个 `package` 开启 `assertion`, 然后在特殊类或 `subpackage` 中关闭其 `assertion`:

```
java -ea:com.example.sorters... -da:com.example.sorters.QuickSort
java -ea:com.example.sorters... -da:com.example.sorters.plugins...
```

如果你比较喜欢冗长的命令行参数, 可以使用 `-enableassertions` 与 `-disableassertions` 来代替 `-ea` 与 `-da`, 以及使用 `-enablesystemassertion` 来代替 `-esa`。

Java 1.4 新增了 `java.lang.ClassLoader` `method`, 用来开启与关闭 `assertion`, 同时通过 `ClassLoader` 来加载类。如果你在程序里使用了一个自定义的类, 又同时想要开启 `assertion`, 你可能会对这些 `method` 感兴趣。

## 使用 assertion

因为 `assertion` 在默认的情况下是关闭的, 而且对程序代码的性能不会有任何的影响, 所以在编写程序时, 你可以充分利用它来记录任何你所想到的东西。若要这么做, 可能会花费一些时间, 但当你做了以后, 你将会发现 `assert` 语句越来越有用。举个例子来说。



假设你用这样的方式来写一个 method，而你知道变量  $x$  不是 0 就是 1，那么在没有 assertion 的情况之下，你可能会写成如下的一个 if 语句：

```
if (x == 0) {  
    ...  
}  
else { // x = 1  
    ...  
}
```

上面的程序代码里的注释是一个非正式的 assertion，它表示你相信在 else 子句的程序主体里， $x$  将永远等于 1。

现在假设你的程序代码是用这样的方式写的： $x$  会出现 0 与 1 以外的其他值，于是和它在一起的注释与假设将不再是合法的，且这会导致程序错误的发生，但不会立刻显现出来。对这种情况的解决方法是将注释转换成 assert statement。程序代码会变成如下：

```
if (x == 0) {  
    ...  
}  
else {  
    assert x == 1 : x // x 必须为 0 或 1  
    ...  
}
```

现在，如果  $x$  拥有了一个非预期值而不知怎的莫名结束后，此时 AssertionError 就会被抛出，它会使得程序错误立刻地就出现且很容易就被找到。此外，在 assert 语句里的第二个表达式（冒号后面的表达式）将非  $x$  所预期的值当成 AssertionError 的“错误信息”。这个信息对用户没有意义，只是提供给你足够的信息，让你不但知道 assertion 失败，还告诉你是什么原因导致它失败。

将它与 switch 语句一起使用会特别的有用。如果你写了一个 switch 语句，且为 switch 表达式的值做了所有可能的假设，但却没有 default 子句。如果你相信不可能还会有其他值时，就可以加入一个 assert 语句来记录与确认这样的事实。例如：

```
switch(x) {  
    case -1: return LESS;  
    case 0: return EQUALS;  
    case 1: return GREATER;  
    default: assert false:x; // 如果 x 不是 -1、0 或 1，就会抛出 AssertionError  
}
```

请注意 assert false; 这个格式一定都会失败。当你认为该语句永远都不会结束时，这个“死胡同 (dead-end)”语句是很有用的。

另一个一般都会使用的assert语句,是测试参数是否都传给了该method(且该method的值都是合法的),这就是我们所知道的强制实行method的先决条件。例如:

```
private static Object[] subArray(Object[] a, int x, int y) {
    assert x <= y : "subArray: x > y";    // 先决条件: x 必须小于等于 y
    // 现在继续创建与返回 a 的子数组 ...
}
```

请注意,这是个专用method。程序员使用了assert语句来说明subArray() method的先决条件与状态,他相信所有调用此专用method的method都有正确的先决条件。这是他可以确定的,因为他可以通过调用subArray()来控制所有的method。可以在测试程序代码时启用assertion来验证他的看法。但一旦程序代码在测试时且assertion是在停用的状态下,则每次该method被调用时就不用为测试这些自变量而感到困扰。请注意,程序员没有使用assert语句来测试自变量a为非null值,且x与y自变量在数组里是合法的索引值。在运行时,Java都会去测试这些本来就有的先决条件,且失败的结果会是uncheckedNullPointerException或ArrayIndexOutOfBoundsException,所以对它们来说,assertion是不必要的。

了解assert语句不适合用在公共method上对其强制实行先决条件是很重要的。公共method可在任何地方被调用,且程序员不能够先断定它会被正常使用。为了使其更加健全,公共API必须明确地测试它的自变量以及强迫它的先决条件每一次都可以被调用到,而不管assertion是否被启用。

一个和使用assert语句有关的是,assert语句会去验证类的不变性。假设你正在创建一个类,用它来表示一个对象列表,而且允许对象被增加或删除,但都会以排序方式维护这个列表。同时你也相信你的实现是正确的,而且插入的method都一定会让这个列表保持顺序,但你想要通过测试来证明它的正确性。此时,你可以写一个method来测试该列表是否确实有顺序性,然后在每个修改列表的method的结尾处使用assert语句来调用该method。例如:

```
public void insert(Object o) {
    ...                // 在这里执行插入
    assert isSorted(); // 在这里确立类的不变性
}
```

在编写必须具有线程安全性(threadsafe)的程序代码时,如果需要使用某个资源,你必须获得该资源的锁(同时使用一个method或语句)。在这种情况下,assert语句会去验证当前的线程是否拥有了它需要的锁:

```
assert Thread.holdsLock(data);
```

Thread.holdsLock() method是在Java 1.4版才新增的,主要是配合assert语句使用。

为了要有效地使用 `assertion`，你必须注意几个规则。首先要记得，有时候你的程序会在启用 `assertion` 时运行，而有时候是在停用 `assertion` 时运行。这代表你应该要特别的小心，不要编写出具有副作用的 `assertion` 表达式。如果你写了一个具副作用的 `assertion` 表达式，则你的程序代码在启用或停用 `assertion` 的状况下的运行是不一样的。当然，对于这个规则，还是有一些例外。例如，如果一个 `method` 包含了两个 `assert` 语句，第一个 `assert` 语句具有副作用而且会影响到第二个 `assertion`；另一个则是在 `assertion` 中使用副作用来确定 `assertion` 是否为启用的状态（你的程序代码实际上并不一定要这样做）：

```
boolean assertions = false; // assertions 是否已启用
assert assertions = true;   // 这个 assert 不会失败，但具有副作用
```

请注意，在 `assert` 语句里的表达式是一个赋值表达式，而不是一个比较表达式。赋值表达式的值一定都是被赋予的值，所以这个表达式计算出来的值一定都会为 `true`，而且 `assertion` 也不会失败。因为这个赋值表达式是 `assert` 语句的一部分，且只有在启用 `assertion` 的状态下，`assertion` 变量才会被设定为 `true`。

在 `assertion` 中除了要避免副作用外，另一个规则就是不需尝试捕获 `AssertionError`（除非你只在最上层捕获，以便于用更具用户友好性的风格来显示错误）。如果 `AssertionError` 被抛出，这表明程序员的其中一个假设没有被拦截。这代表程序代码在使用上超出了参数原先被设计的范围。总之，没有一种实际可行的方式可以从 `AssertionError` 中恢复正常，你不该尝试去捕获它。

## Method

方法（`method`）是由一组 Java 语句所构成的命名序列（`named sequence`），它可以被 Java 程序代码调用。当 `method` 被调用时，可以有零个或更多的值（也就是自变量）传递给它。`method` 会执行一些运算而且可能会有返回值。就如本章的“表达式与运算符”一节中所描述的，`method` 调用（`method invocation`）是一个能被 Java 解释器计算出值的表达式。因为方法调用具有副作用，所以它也可以被当作表达式语句来使用。

### 定义方法（defining method）

你已经知道如何去定义方法的主体了，它只是一些被大括号括起来的语句。与 `method` 相关的较有趣的项目是签名（`signature`）（注 5）。签名指定了以下项目：

---

注 5： 在 Java 语言的规范说明书里，专有名词“`signature`”有一个技术上的意义，它与在这儿所使用的 `signature` 有些微的不同。本书使用了 `method signature` 较不正式的定义。



- 方法的名称
- 方法所使用的每一个参数的名称、类型与顺序
- 方法返回值的类型
- 此方法所能抛出的查核异常（签名也能列出未查核异常，但这并非必要的）
- 方法的修饰符，用来提供其他有关于方法的信息

method 签名定义了所有在你调用该 method 之前所必须知道的信息，它是该 method 的详细规范说明书 (specification)，并定义了该 method 的 API。为了能够编写 Java 程序，你必须知道应该如何去定义你自己的 method，因为每个 method 都必须有一个 method 签名。

method 签名形式如下：

```
modifiers type name ( paramlist ) [ throws exceptions ]
```

签名 (method 规范说明书) 后紧跟着的是 method 的主体 (method 的实现)，那是由 Java 语句所组成的，并用大括号括起来。如果 method 是抽象的 (请参阅第三章)，实现部分会被省略，method 的主体部分就只有一个分号而已。在 Java 5.0 和之后的版本中，泛型方法 (generic method) 的签名也可能会包括变量类型声明。泛型方法与变量类型将在第四章中讨论。

以下是一些 method 定义的范例，是由签名开始且后面接着 method 主体：

```
// 此 method 会传入字符串数组且没有返回值
// 所有的 Java 程序都有一个 main 进入点与签名
public static void main(String[] args) {
    if (args.length > 0) System.out.println("Hello " + args[0]);
    else System.out.println("Hello world");
}

// 此 method 会传入两个 double 自变量，同时会返回一个 double 类型的值
static double distanceFromOrigin(double x, double y) {
    return Math.sqrt(x*x + y*y);
}

// 这是一个抽象的 method，这代表它没有主体
// 请注意，当它被调用时，可能会抛出异常
protected abstract String readText(File f, String encoding)
    throws FileNotFoundException, UnsupportedEncodingException;
```

*modifiers* 是零个或多个修饰符关键字，它们之间使用空格来分开。例如，在声明 method 时可能会加上 `public` 与 `static` 修饰符来声明。在下一个章节中会对被允许的修饰符及其意义做说明。

method 签名中的 *type* 指明了 method 的返回值类型。如果该方法没有返回值，*type* 就必

须为 `void`。如果该方法声明了一个非 `void` 的返回值类型，则在其代码块里必须包含一个 `return` 语句，且返回值的类型要和所声明的类型一样。

构造函数是个特殊的 `method`，用来初始化新创建的对象。就如我们将在第三章中看到的，除了构造函数的签名不包含 `type` 规范说明外，它的定义与 `method` 基本上是相同的。

`method` 的名称是紧接在它的修饰符的规范说明与类型之后的。方法名称 (`method name`) 就跟变量名称一样，都是 Java 的标识符，而且它跟所有的 Java 标识符一样，都可以使用 Unicode 字符集里的所有字符。使用相同的名称去定义一个以上的 `method` 是合法的，有时候甚至相当有用，只要每个方法的参数列表不同就可以了。将多个 `method` 定义成相同的名称称作方法重载 (`method overloading`)。我们常看到的 `System.out.println()` `method` 便是一个方法重载的例子，有一个以此为名称的 `method` 会列出字符串，也有其他相同名称的 `method` 会列出不同类型的值，Java 解释器会依据传递给此方法的自变量类型来决定调用哪一个 `method`。

当你在定义一个 `method` 时，`method` 的名称后面就紧跟着 `method` 的参数列表，而参数列表要使用小括号将它们给括起来。参数列表可定义零个以上传递给该 `method` 的自变量。自变量定义的格式为每组参数必须包含该参数的类型与名称，同时各组之间使用逗号来分隔（如果有多个参数时）。当一个 `method` 被调用时，传递给它的自变量值必须符合在 `method` 签名中所设定的参数的数目、类型相符。传递给 `method` 的值，其类型并不一定要跟签名所设定的相同，但它们必须不用强制转换便可以转换成签名所指定的类型才可以。C 与 C++ 的程序员必须特别注意的是，当 Java 的 `method` 不需要任何的自变量时，则它的参数列表写法为 `()`，而不是 `(void)`。

在 Java 5.0 或以上的版本中，定义与调用可变动其自变量数目的 `method` 是有可能的，这在语法中称作 `varargs`，这将在稍后的章节做详尽的介绍。

`method` 签名的最后一部分就是 `throws` 子句，`method` 会通过它来抛出已查核的异常列表。已查核异常 (`checked exception`) 是必须要被列于 `method` 的 `throws` 子句中的异常类的种类，这样才可以将它们给抛出。如果 `method` 使用 `throws` 语句来抛出一个已查核异常，或者调用其他会抛出已查核异常的 `method` 但是并不会捕获或处理该异常，则该方法必须声明它会抛出该异常。如果 `method` 会抛出一个或一个以上的已查核异常，则它必须被指明出来，也就是把 `throws` 关键字放在参数列表之后并输入它所会抛出的异常类。如果 `method` 不会抛出任何异常，便不需要用到 `throws` 关键字。如果 `method` 抛出了超过一种类型的异常时，则必须使用逗号将这些异常类加以分隔。其他要注意的事项就不多了。

## method 修饰符

method 的修饰符是由零个或零个以上的修饰符关键字所组成的, 如 `public`、`static` 或 `abstract`。这里列出了修饰符与它的意义。请注意, 在 Java 5.0 及以上的版本里, `annotation` (如 `@Override`、`@Deprecated` 与 `@SuppressWarnings`) 可被视为修饰符, 而且可以和以下的修饰符混合在一起使用。其实任何人都可以定义新的 `annotation` 类型, 所以要列出所有的 `method annotation` 是不可能的, 第四章将会介绍更多关于 `annotation` 的信息。

### `abstract`

该类含有未实现且抽象化 (`abstract`) 的 `method`, 同时该 `method` 没有程序主体。包含此 `abstract method` 的类必须被声明为 `abstract`。这样的类是不完整的, 而且无法被实现 (请参阅第三章)。

### `final`

`final method` 不可以被子类覆盖或隐藏, 它可以达成对正常 `method` 来说不可能的编译器最佳化。所有的 `private method` 其实都是 `final`, 就如同类中所有的 `method` 都会被声明为 `final` 一样。

### `native`

`native` 修饰符用来表示该 `method` 是以 “`native`” 语言 (如 C) 来实现的, 同时也被提供给 Java 程序。就像 `abstract method` 一样, `native method` 没有主体, 在它后面紧跟着的就是一个分号。

当 Java 第一次被发布时, 由于效率的原因而有时候使用 `native method`。现今已经不再需要使用它了, 取而代之的是 `native method` 被当成 Java 程序与现有 C 或 C++ 所编写的程序库之间的接口。`native method` 具平台依赖性, 用来链接 Java 类实现的程序所声明的 `method` 依赖于 JVM 的实现。`native method` 的内容未包括于本书中。

### `public`、`protected`、`private`

这些修饰符指明了该 `method` 是否可以在定义它的类之外被使用以及该 `method` 可在何处被使用。这些重要的修饰符将在第三章中介绍。

### `static`

一个被声明为 `static` 的 `method` 是一个与它自己的类相关的类方法 (`class method`), 而不是该类的实例。第三章将会更详细地介绍。

### `strictfp`

被声明为 `strictfp` 的 `method` 必须使用 32 或 64 位的浮点格式来执行浮点运算, 不能利用其他额外的该平台所能支持的浮点硬件可用的指数位。在这个命名不恰当、极少使用的修饰符中的 “`fp`” 代表 “`floating point`”。



synchronized

synchronized 修饰符可以让 method 具有线程安全性。在一个线程在运行 synchronized method 之前,它必须获得一个该方法的类(针对 static method)或相关的类实例(针对非 static method)的锁。这个锁可用来防止两个线程同时执行一个 method。synchronized 修饰符是一个方法的实现细节 (implementation detail, 因为 method 可以使用其他方式使它们自己具线程安全性),而且它不是正式的 method 规范说明书或 API 的一部分。好的说明文件会详细说明该 method 是否具线程安全性,当在运行多线程程序时,不应该依赖是否看到了 synchronized 关键字。

## 声明已查核异常

在讨论到 throw 语句时,我们说异常是 Throwable 对象且该异常又有两个主要的类,分别是 Error 与 Exception 的子类。除了区分为 Error 与 Exception 这两个类,Java 的异常处理机制也可以区别已查核与未查核的异常。任何属于 Error 的异常对象都是未查核的,而属于 Exception 的异常对象除了 java.lang.RuntimeException 之外都是已查核的 (RuntimeException 是 Exception 的子类)。

要区别已查核与未查核异常必须依据异常被抛出的情况。实际上,任何时候任何 method 都可能会抛出未查核的异常。例如,没有方法可以预知 OutOfMemoryError;又如,如果传入了一个不合法的 null 自变量值,则任何使用对象或数组的 method 都会抛出 NullPointerException。然而,已查核异常只会发生在某些特殊且定义明确的情况下。如果你试着要从某个文件里读取数据,就必须考虑到如果该文件不存在的话,FileNotFoundException 将会被抛出。

Java 对于已查核与未查核异常有着不同的处理规则。如果你写的 method 会抛出已查核异常,就必须在 method 的签名里使用 throws 子句来声明该异常。将这些类型的异常称作已查核异常的原因是,Java 解释器会检查你是否在 method 的签名中声明了它们,如果你没有声明,编译器就会产生一个编译错误。

即使你从来没有使用过 throws 来抛出异常,但有时候你还是必须使用 throws 子句来声明异常。如果你的 method 调用了一个会抛出已查核异常的 method,则你必须加入一段异常处理程序代码来处理该异常,或使用 throws 来声明你的方法可以抛出该异常。例如,以下的 method 会从一个文件中读入第一行文字,它使用的 method 可以抛出 java.io.IOException 对象中的各种类型,所以它在声明时使用了 throws 子句:

```
public static String readFirstLine(String filename) throws IOException {  
    BufferedReader in = new BufferedReader(new FileReader(filename));  
    String firstline = in.readLine();  
}
```

```
        in.close();
        return firstline;
    }
```

要如何知道你所调用的method是否可以抛出已查核异常呢？你可以通过查看该method的签名得知；或是，在没有这么做的情况下，如果你所调用的method中包含了必须加以处理或声明的异常，Java编译器会直接告诉你（通过报告编译错误）。

## 不定长度的自变量列表

在Java 5.0或以上的版本中，method可被声明为可接受不定数量的自变量，而且也可以被不定数量的自变量所调用。这种method就是我们所说的 *varargs* method。新的 `System.out.printf()` method 及与其相关的 `String` 的 `format()` method 和 `java.util.Formatter` 都使用了 *varargs*。相似但不相关的 `java.text.MessageFormat` 的 `format()` method 已经被转换为使用 *varargs*，就像 `java.lang.reflect.Refelction` API 的一些重要 method 一样。

一个不定长度的自变量列表的声明是在method里使用省略号(...)来表示该method可以接受不定长度的自变量，并指出其自变量可以被重复零次或多次。例如：

```
public static int max(int first, int... rest) {
    int max = first;
    for(int i: rest) {
        if (i > max) max = i;
    }
    return max;
}
```

上述的 `max()` method 声明了两个自变量，第一个是正常的 `int` 值，第二个则可以被重复零次或更多次。以下使用 `max()` 的方式都是合法的：

```
max(0)
max(1, 2)
max(16, 8, 4, 2, 1)
```

就如你从 `max()` 主体中的 `for/in` 语句所看到的，第二个自变量可被视为由 `int` 值组成的数组。*varargs* method 完全都是由编译器处理。对Java解释器来说，`max()` method 和以下的程序没有两样：

```
public static int max(int first, int[] rest) { /* 省略程序主体 */ }
```

若要将 *varargs* 签名转换成“真实的”签名，只要用 `[]` 取代 `...` 就可以了。记住，只有一个省略号(...)可以出现在参数列表里，而且它也只能够出现在参数列表的最后一个参数上。

因为 varargs method 会将预期的自变量数组编译进该 method 中，所以在调用这些编译好的 method 时，该 method 已经将数组的建立与初始化包含在里面了。因此，在调用 `max(1, 2, 3)` 后其会被编译为：

```
max(1, new int[] { 2, 3 })
```

如果你有一个自变量为数组的 method，那么将该自变量数组传递给 method 的做法是完全合法的，而且是非常特别的写法。你可以将任何的... 自变量像一个数组那样声明。然而，相反的做法却不成立：当该 method 使用... 来声明其为 varargs method 时，你就只可以使用这样的语法。

Varargs method 和 Java 5.0 的新功能 autoboxing 之间的交互特别良好（请看稍后关于“Boxing 与 Unboxing 的转换”一节）。一个 method，当它的自变量是一个拥有 Object... 不定长度的自变量列表时，它可以使用任何一种引用类型的自变量，因为所有的对象与数组都是 Object 的子类。而且，autoboxing 也允许你可以使用基本类型值来调用 method：编译器会将它们给封装成 wrapper 对象，就像编译器为 method 建立放置自变量的 Object[] 一样。在本章一开始，`printf()` 与 `format()` method 就提到它们可以被声明为 Object... 参数。

method 与 Object... 参数产生了一个很特别的情况，实际上它很少会出现，但一旦你学会了，将能把你所了解的 varargs 融会贯通。varargs method 可以使用数组类型的自变量或任何数目的元素类型自变量。当一个 method 的自变量被声明为 Object... 时，你可以传递一个自变量的 Object[]，或零个或以上的单独 Object 自变量给它，但其实每一个 Object[] 也是一个 Object。如果想要传递一个作为单独对象自变量 (single object argument) 的 Object[] 给 method 时，该怎么做呢？请参考以下使用 `printf()` method 的程序代码：

```
import static java.lang.System.out; // 现在 out 就是指 System.out

// 在这里我们调用了使用单独的 Object 自变量的 varargs method
// 注意，在这里使用了 autoboxing 将基本类型转换为 wrapper 对象
out.printf("%d %d %d\n", 1, 2, 3);

// 这行是在做相同的事情，但会传递数组类型自变量
// 该自变量数组已经建立好了
Object[] args = new Object[] { 1, 2, 3 };
out.printf("%d %d %d\n", args);

// 现在看一下以下的 Object[]，
// 我们希望传递的是单独自变量，而不是数组类型的两个自变量
Object[] arg = new Object[] { "hello", "world" };
// 这两行做同样的事情：输出 "hello"，但并不是我们所想要的
out.printf("%s\n", "hello", "world");
out.printf("%s\n", arg);
```



```
// 如果我们想要令 arg 被视为是单独的 Object 自变量，
// 我们就需要将 arg 作为数组的元素传递给它
out.printf("%s\n", new Object[] { arg });

// 一个比较简单的方法是说服编译器自己去创建数组
// 我们使用强制转换来将 arg 转换为单独的 Object 自变量，而不是数组
out.printf("%s\n", (Object) arg);
```

## 协变返回类型 (Covariant Return Type)

它是泛型 (generic type) 的一部分，现在 Java 5.0 也支持协变返回。也就是覆盖 method (overriding method) (注 6) 可以将覆盖该 method 的返回类型给缩小。这从以下的范例可以很清楚地看到：

```
class Point2D { int x, y; }
class Point3D extends Point2D { int z; }

class Event2D {
    public Point2D getLocation() { return new Point2D(); }
}

class Event3D extends Event2D {
    @Override public Point3D getLocation() { return new Point3D(); }
}
```

上述的程序代码定义了四个类：一个是二维空间的点，一个是三维空间的点，事件对象 (event object) 则表示在二维空间与三维空间中的事件。每一个事件类都有一个 getLocation() method。Event2D method 返回一个 Point2D 对象。Event3D 是 Event2D 的一个子类，同时它也覆盖了 getLocation()。很明显，可以看到该 method 返回一个 Point3D 类型的值，因为每一个 Point3D 对象也可以说是 Point2D 对象，这么做是非常合理的。在 Java 5.0 以前没有提供如此简单的功能。

在 Java 1.4 及以前的版本里，覆盖方法 (overriding method) 的返回类型必须与覆盖的 method 的类型一样。为了在 Java 1.4 中能编译，Event3D.getLocation() method 的返回类型将会被修改为 Point2D。当然，它仍然会返回 Point3D 对象，但调用者会将返回值从 Point2D 强制转换为 Point3D。

程序范例中的 @Override 是一个 annotation，这在第四章中会做说明。这是一个编译时的 assertion，且其 method 会改写某些东西，如果 assertion 失败了，则编译器会产生编译错误。

注 6: method overriding 与 method overloading 是不相同的，这在稍早的章节中就介绍过了。method overriding 包含了子类，在第三章会做说明。如果你对于这个概念并不是那么熟悉的话，现在可以先跳过本小节，之后再回来继续阅读。

## 类与对象

现在我们已经介绍过运算符、表达式、语句与方法了，最后我们要来谈谈类。类 (class) 是由许多保存数据值的字段以及在这些数据值上进行操作的method所组成的命名集合。类是Java所支持的五种引用类型之一，但它是最重要的一种类型。第三章全部是在介绍类。之所以在这里介绍，是因为它是继method之后另一个较高级的语法，同时本章后面的章节的内容都需要我们对class有一些基本的概念。

class如此重要的另一个原因是：每一个类都定义了一个新的数据类型。例如：你可能会定义一个名为Point的类来表示二维直角坐标系统上的数据点。此类可以定义字段 (double类型) 来表示点的X与Y坐标，也可以定义方法来操作该点。Point类便是一个新的数据类型。

当在讨论数据类型的时候，将数据类型本身与该数据类型所代表的数值加以分隔是相当重要的一件事。char是一个数据类型，它表示Unicode字符，但一个char值所代表的则是某个特定字符。类是一种数据类型，该数据类型的值称作对象 (object)。我们使用类这个名称是因为每个类都定义了对应的类型 (或种类、类型)。Point类是一个用来表示X、Y坐标平面上的点的数据类型，而Point对象则代表某个X、Y坐标平面上的点。或许你已经体会到，类与它们的对象的关系是如此紧密，在接下来的几节中，我们会再详加讨论。

## 定义一个类

下面是对我们讨论的point类的一个可能定义：

```
/** 表示一个直角坐标点 (x,y) */
public class Point {
    public double x, y;                // 该点的坐标
    public Point(double x, double y) { // 一个用来初始化字段的构造函数
        this.x = x; this.y = y;
    }

    public double distanceFromOrigin() { // 操作x与y字段的method
        return Math.sqrt(x*x + y*y);
    }
}
```

此类的定义是存储在Point.java文件中，同时编译后的结果存储于Point.class文件中，Java程序与其他的类可以经由Point.class来使用此类。此类的定义相当的完整，但目前你并不需要急于了解所有的细节。第三章的主要内容即针对类的定义，你可以到那时再慢慢了解。

但请记住，你并不需要在Java程序中定义所有你会用到的类，Java平台已包含了上千个预先定义好的类供你使用。

## 创建一个对象

现在我们已经定义好Point类，并将它作为一个新的数据类型，可以使用以下的方法去声明一个拥有Point对象的变量：

```
Point p;
```

声明一个拥有Point对象的变量并不会创建该对象，为了要真正创建该对象，你必须使用new运算符。此关键字的后面紧跟着的是该对象的类（即它的类型）以及用一对小括号括起来的自变量列表。这些自变量是要传递给类的构造函数方法，由它来初始化该新对象的字段值：

```
// 创建一个代表(2, -3.5)的Point对象，
// 声明一个变量p，并将新创建的Point对象存在变量p中
Point p = new Point(2.0, -3.5);

// 创建其他对象
Date d = new Date();           // 一个用来表示当前时间的Date对象
Set words = new HashSet();     // 一个拥有一组对象的HashSet对象
```

new关键字是Java中最常被用来创建对象的方法，当然还有其他的方法也具有相同的功能。首先，有一些类十分的重要，因此Java特别为它们定义较为特殊的语法来创建这些类型的对象（我们将在稍后讨论到）。第二，Java支持动态加载机制，让程序可以动态地加载类并创建这些类的对象，这些机制都是由java.lang.Class与java.lang.reflect.Constructor中的newInstance() method来完成的。最后，对象也可以通过反序列化（deserialize）来创建。也就是说，对象会将它的状态保存或者说序列化（serialize）于文件中，并可使用java.io.ObjectInputStream类来重新创建。

## 使用对象

现在我们已经学会了如何定义类与创建对象，接下来我们必须了解当使用这些对象时，必须遵循哪些Java语法。类定义了一些字段与方法，且每一个对象都有一份属于它自己的字段与方法的副本（copy），我们使用点号字符（.）来访问对象中的这些字段与方法。例如：

```
Point p = new Point(2, 3);           // 创建一个对象
double x = p.x;                      // 读取对象中的一个字段
p.y = p.x * p.x;                     // 设定该字段的值
double d = p.distanceFromOrigin();   // 访问该对象的一个method
```



此语法是Java面向对象程序设计中最重要的一部分，因此你将会常常看到这样的用法。请注意 `p.distanceFromOrigin()` 表达式，它将会告诉Java编译器去寻找一个名为 `distanceFromOrigin()` 的 method，并使用该方法来对对象 `p` 的字段执行一些计算操作。我们将会在第三章做详细的介绍。

## 对象直接量

在我们讨论基本数据类型时，我们看到每一个基本数据类型都有一个直接量语法，并逐字将类型的值加到程序的文本中。Java也为特殊的引用类型定义了直接量语法，以下将对其做一介绍：

### 字符串直接量

`String` 类，它将文字以字符串的方式加以表现，因为程序通常都是使用文字来与用户沟通的，因此操作文字字符串的能力在任何的程序语言中都是相当重要的。在某些语言中，字符串是属于基本数据类型，就好像整数与字符一样。在Java中，字符串则是对象，用来表示文字的数据类型是 `String` 类。

因为字符串是一个相当基本的数据类型，Java让你在程序中使用双引号 (") 字符来置放字符串，例如：

```
String name = "David";
System.out.println("Hello, " + name);
```

请不要将双引号字符与单引号（或撇号）字符相混淆，双引号字符是用来括住字符串，而单引号字符则是用来括住字符的。字符串直接量 (string literal) 可以包含任何 `char` 直接量允许的转义序列（请参阅表 2-2）。当你在字符串中加入双引号字符时，使用转义序列特别有用，例如：

```
String story = "\t\"How can you stand it?\" he asked sarcastically.\n\";
```

字符串直接量不可以包含注释，而且只能有一行。Java不支持任何可以将两行当成一行对待的延长字符。如果你需要表示一个长度大于一行所能容纳的字符串时，你可以将它们打散成独立的字符串直接量，并使用 `+` 运算符把它们连接起来，例如：

```
String s = "This is a test of the          // 这是不合法的，
            emergency broadcast system";   // 字符串直接量不可以跨越好几行

String s = "This is a test of the " +      // 可以改成这样
            "emergency broadcast system";
```

这样的字符串直接量连接会在程序编译时进行，而不会等到运行时才进行，因此你不用担心会影响到程序运行的效率。

## 类型直接量 (Type literal)

第二种支持特殊对象直接量语法的是Class类。Class类的实例是一个Java数据类型。在Java程序里加入一个Class对象,可在任何数据类型的名称后面加上.class,例如:

```
Class typeInt = int.class;
Class typeIntArray = int[].class;
Class typePoint = Point.class;
```

## null 引用

null关键字是一个特殊的直接量,它表示引用到一个不存在的东西或是没有引用到任何东西。null值是独一无二的,因为它是每个引用类型的成员,你可以将null指定给任何一种引用类型的变量。例如:

```
String s = null;
Point p = null;
```

## 数组

数组是对象中很特别的一种类型,它可以拥有零个以上的基本数据类型值或对象。这些值被数组元素所拥有,且它们是被它们所在的位置或索引引用到的未被命名的变量。数组的类型所具有的特征和它的元素类型一样,且数组里的所有元素都必须是同一种类型。

数组元素的编号从零开始,元素的合法索引值范围是由零开始到元素个数减一。例如,索引值为1的数组元素是数组中的第二个元素。数组元素的个数就是该数组的长度,当数组被创建好以后,它的长度就已经被确定好了且不能变动。

数组元素的类型可以是任何合法的Java类型,包括数组类型,也就是说Java支持了由数组组成的数组,它提供了多维数组类型,但Java不支持矩阵形式的多维数组。

## 数组类型

数组类型是一种和class一样的引用类型。数组的实例和class的实例一样也是对象(注7),但不同于class的是,数组类型不需被定义,只需在元素类型的后面加上一对中括号即可。以下的程序代码是声明三个数组类型的变量的范例:

---

注7: 在讨论数组时,有个术语上的难题。与在处理类和它们的实例时不同,我们同时对数组类型和数组实例使用“数组”这个术语。实际上,通常通过上下文就能清楚看出被讨论的对象是类型或是值。

```
byte b;                // byte 是基本数据类型
byte[] arrayOfBytes;    // byte[] 是一个数组类型: byte 的数组
byte[][] arrayOfArrayOfBytes; // byte[][] 是另一种类型: byte[] 的数组
String[] points;        // String[] 是一个 String 对象的数组
```

数组长度不是数组类型的一部分。例如,声明一个method,并期望它的数组恰好是四个int值,这是不可能的。如果method的参数是由int[]类型所构成的,则调用者就可以传递拥有任意个数的元素(包括零个元素)之数组给它。

数组类型不是class,但数组的实例却是一个对象,也就是数组继承了java.lang.Object的method。数组实现了Cloneable接口并改写了clone() method以保证数组无论如何都可以被复制且clone()绝对不会抛出CloneNotSupportedException。数组也可以实现Serializable,所以如果数组元素的类型可以被序列化(serialized)的话,则任何的数组都可以被序列化。最后,所有的数组都有一个名称为length的public final int字段,它说明了数组中元素的个数。

### 数组类型的扩大转换

因为数组可以扩展(extend)Object并实现Cloneable及Serializable接口,所以任何数组类型都可以被扩大(widen)成这三种类型。但某些数组类型也可以被扩大成其他的数组类型。如果数组元素的类型是一个引用类型T,而T可以被指定给类型S,则数组类型T[]也可被指定给数组类型S[]。请注意,基本数据类型的数组类型是不可以扩大转换的。以下的程序代码是合法的数组扩大转换范例:

```
String[] arrayOfStrings;    // 创建字符串数组
int[][] arrayOfArrayOfInt;  // 创建int二维数组
// String 可被指定给 Object, 所以 String[] 可被指定给 Object[]
Object[] oa = arrayOfStrings;
// String 实现 Comparable, 所以 String[] 可以被视为 Comparable[]
Comparable[] ca = arrayOfStrings;
// int[] 是一个 Object, 所以 int[][] 可被指定给 Object[]
Object[] oa2 = arrayOfArrayOfInt;
// 所有的数组都是 Cloneable, serializable 的对象
Object o = arrayOfStrings;
Cloneable c = arrayOfArrayOfInt;
Serializable s = arrayOfArrayOfInt[0];
```

具有将数组类型扩大为其他数组类型的能力,即表示数组类型的编译时间和运行时间是不一样的。编译器必须经常在将引用值存储于数组元素之前插入运行时检查,以确保该值的运行时类型与数组元素的运行时类型是一致的。如果运行时的检查失败了,ArrayStoreException就会被抛出。



## 与 C 兼容的语法

如前面所看到的，数组类型是被放置在[]字符之后。为了与 C 及 C++ 兼容，Java 为变量声明提供了另一个语法选择，中括号[]除了可被放在数组元素类型后面之外，它也可被放在变量名称的后面。这适用于局部变量、字段与 method 参数。例如：

```
// 此行声明了类型为 int、int[] 与 int[][] 的局部变量
int justOne, arrayOfThem[], arrayOfArrays[][];

// 以下三行声明了相同数组类型的字段：
public String[][] aas1;    // 较好的 Java 语法
public String aas2[][];    // C 语法
public String[] aas3[];    // 令人困惑的混合语法

// 此 method 签名包含了两个相同类型的参数
public static double dotProduct(double[] x, double y[]) { ... }
```

这样的语法总是让人相当的困惑，其实并不建议你使用。

## 创建与初始化数组

在 Java 中你必须使用 new 关键字来创建一个数组，就好像你在创建对象时一样。数组类型没有构造函数，但在创建数组时你必须指定它的长度，在中括号内使用正整数值来指明你想要的数组大小：

```
byte[] buffer = new byte[1024];    // 创建一个拥有 1024 字节的 byte 数组
String[] lines = new String[50];    // 创建一个长度为 50 的 String 数组
```

当你使用以上的语法创建数组时，数组中的每一个元素都会自动被初始化为相同的默认值，在 class 的字段中也是这样的用法：即对布尔值来说，该默认值为 false，对 char 来说为 '\u0000'，对整数来说为 0，对浮点数来说就是 0.0，而对引用类型来说则为 null。

数组表达式也可以被用于创建或初始化一个多维的矩形数组，这种语法就更复杂了，在稍后的章节将会说明。

## 数组初始化程序

若要在一行表达式中创建数组及其元素做初始化，通常都会省略掉数组长度，并在中括号后紧接着由大括号括住，并使用由逗号将初始值分开的表达式列表。当然，每一个表达式的类型必须是数组元素所指定的类型。数组的长度就等于表达式的数量。在表达式的最后加上一个逗号是合法的，但没有必要这样做。例如：

```
String[] greetings = new String[] { "Hello", "Hi", "Howdy" };
int[] smallPrimes = new int[] { 2, 3, 5, 7, 11, 13, 17, 19, };
```

这样的语法允许数组被创建并初始化，且不需要将它赋值给变量。就某种意义来说，这些数组创建表达式是属于匿名数组直接量（anonymous array literal）。这里有些例子：

```
// 调用一个method，同时传递一个拥有两个字符串的匿名数组直接量
String response = askQuestion("Do you want to quit?",
                               new String[] { "Yes", "No" });

// 调用另一个是匿名数组（即匿名对象，anonymous object）的method
double d = computeAreaOfTriangle(new Point[] { new Point(1,2),
                                                new Point(3,4),
                                                new Point(3,2) });
```

当数组初始化是变量声明的一部分时，你可以省略掉new关键字与元素类型，并将大括号内的数组元素列出：

```
String[] greetings = { "Hello", "Hi", "Howdy" };
int[] powersOfTwo = {1, 2, 4, 8, 16, 32, 64, 128};
```

Java虚拟机的结构不支持任何有效的数组初始化。换句话说，数组直接量会在程序运行时被创建与初始化，而不是在程序被编译时。考虑以下的数组直接量：

```
int[] perfectNumbers = {6, 28};
```

它会被编译成与以下代码相等的Java字节码：

```
int[] perfectNumbers = new int[2];
perfectNumbers[0] = 6;
perfectNumbers[1] = 28;
```

如果你想要为大量的数组初始化，则在程序中加入数组值之前必须要多加考虑。这并不是个好主意，因为Java编译器必须建立许多的字节码来初始化数组，如此就需要更多的空间来存放这些数据。在这样的情况之下，我们建议可将这些数据存放在另外的文件中，并在要运行时再读进程序中。

但是，Java程序代码在运行时进行所有数组的初始化导致了一个重要的必然结果。这代表了数组初始化表达式可以在运行时再计算，不必是编译期解决的常量表达式。例如：

```
Point[] points = { circle1.getCenterPoint(), circle2.getCenterPoint() };
```

## 使用数组

一旦数组被创建好后，你就可以开始使用它了。以下章节会说明数组元素的访问方式及数组的使用方法，如进行数组元素迭代处理（iterating）的操作及数组的复制等。

## 访问数组元素

数组中的元素是可变的。当数组元素在表达式中出现时，它会计算出该元素的值；当数组元素出现在赋值操作数的左边时，则新的值就会存储于该元素中。和正常的变量不一样的是，数组元素是没有名字的，只有一个编号而已。数组元素是使用方括号来被访问，如果 `a` 是一个可以计算出值并供数组访问的表达式，则可以使用 `a[i]` 来指明该元素，其中 `i` 是一个整数直接量或可以计算出 `int` 值的表达式。例如：

```
String[] responses = new String[2];    // 创建拥有两个字符串的数组
responses[0] = "Yes";                  // 设定数组的第一个元素
responses[1] = "No";                   // 设定数组的第二个元素

// 现在读取这些数组元素
System.out.println(question + " (" + responses[0] + "/" +
                           responses[1] + " ): ");

// 同时拥有数组引用与数组索引的复杂表达式
double datum = data.getMatrix()[data.row()*data.numColumns() + data.column()];
```

数组索引表达式必须是 `int` 类型，或是可以被扩大为 `int` 的类型：`byte`、`short` 或 `char` 类型。很明显，数组的索引若是使用布尔、浮点或双精度浮点类型将是不合法的。还应记住数组的 `length` 字段就是一个 `int` 值，且该数组的长度不可超过 `Integer.MAX_VALUE` 元素里的值。拥有表达式的数组索引值若为 `long` 类型，则在编译时也会发生错误，即使表达式的值的范围在运行时在 `int` 之内。

## 数组界限

还记得数组的第一个元素是 `a[0]`，第二个元素是 `a[1]`，而最后一个元素是 `a[a.length-1]` 吧！如果你习惯的语言所使用的数组是从 1 开始的，那么你可能要慢慢习惯这个以 0 为基础的数组了。

在数组里最常见的程序错误是所使用的索引值太小（负数索引）或太大（大于或是等于数组的长度）。在某些像 C 或 C++ 的程序语言中，常常会发生所写的程序代码使用了小于数组最小值的元素或超过数组最末端的元素而产生了无法预期的结果。这样的程序错误并不是在每次错误情形发生时都会被捕获，有可能过了一阵子之后才会被发现。但在 Java 程序代码中编写一个不完整的数组索引的话，Java 解释器在运行时去检查每一个数组的访问，以保证都是其所预期的结果。如果数组索引值太小或太大，Java 解释器就会立刻抛出 `ArrayIndexOutOfBoundsException` 异常。

## 迭代处理数组

循环的一般写法是通过数组元素迭代处理该数组，这样做是为了完成一些运算。这是 `for` 循环最典型的做法。以下的范例是计算整数数组的总和：



```
int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
int sumOfPrimes = 0;
for(int i = 0; i < primes.length; i++)
    sumOfPrimes += primes[i];
```

这个 for 循环符合语法的结构，你经常会看到。

在 Java 5.0 或以上的版本中，数组也可以使用 for/in 循环来迭代。上述计算总和的程序可以被改写为以下更简洁的样式：

```
for(int p : primes) sumOfPrimes += p;
```

## 复制数组

所有的数组都可以实现 Cloneable 接口，且任何的数组都可以通过使用 clone() method 被复制。请注意，你必须将 clone() 的返回值强制转换为适当的数组类型，如此，数组的 clone() method 就不会抛出 CloneNotSupportedException；

```
int[] data = { 1, 2, 3 };
int[] copy = (int[]) data.clone();
```

clone() method 可以为数组制作浅层 (shallow) 复制。如果数组的元素类型是一个引用类型时，只有该数组中的引用会被复制，而这些被引用对象则不会被复制。因为这是浅层复制，所以任何的数组都可以被复制，即使该数组元素的类型没有自己的 Cloneable。

有时候你会想要将一个已经存在的数组中的元素复制到另一个已经存在的数组中，System.arraycopy() method 就是被设计来做这样的操作的，同时你可以将它想成是 Java VM 在硬件层使用了快速局部复制运算来完成复制操作。

arraycopy() 是一个很容易被了解的函数，但它在使用上有些困难，因为它有五个参数要记；第一个会传递将要被复制元素的源数组，第二个会传递该数组起始元素的索引值，第三个与第四个则分别传递目的地数组与其索引值，第五个参数则是明确地指定有多少个元素将被复制。

即使在相同的数组里有部分重叠的副本时，arraycopy() 还是可以正常运行。例如，你要“删除”数组 a 中索引值为 0 的元素或是你想要在索引值 1 到 n 之间平移元素以便它们占住索引值 0 到 n-1 时，可以用以下的方法：

```
System.arraycopy(a, 1, a, 0, n);
```

## 数组公用程序

`java.util.Arrays`类提供了可以和数组一起运行的一些静态公用程序method。对于每一个基本数据类型的数组的版本与其他对象的数组的版本，这些method大部分都是重度重载的（overloaded）。`Sort()`与`binarySearch()` method对于数组的排序与搜索特别的有用；`equals()` method可以让你对两个数组的内容作比较；而当你想将一个数组的内容转为字符串时，例如在调试或记录输出信息时，`Arrays.toString()` method很有用。

在Java 5.0中，`Arrays` class包含了`deepEquals()`、`deepHashCode()`与`deepToString()` method，这些method都可以在多维数组中使用。

## 多维数组

就如我们之前所看到的，数组类型是由元素类型后再接着的一对中括号表示，`char`数组可以写成`char[]`，`char`数组的数组则可以写成`char[][]`。当一个数组元素本身就是一个数组时，我们就称此数组为多维数组。如果要使用多维数组，就必须了解更多补充的细节。

假设你想要使用一个多维数组来表示一个乘法表：

```
int[][] products; // 一个乘法表
```

每一对中括号代表了一个维度，所以这是一个二维数组。为了要在此二维数组里访问一个`int`元素，你必须使用两个索引值，也就是每一个维度要指定一个索引值。假设此数组在一开始时便被初始化成乘法表，则存储在每个`int`元素中的值便是两个索引值的乘积。也就是说`products[2][4]`等于8，`products[3][7]`等于21。

为了要创建一个新的多维数组，你必须使用`new`关键字并指定每一个维度的大小。例如：

```
int[][] products = new int[10][10];
```

在某些语言中，像这样的数组会被创建成一个由100个`int`值所组成的代码块。但Java却不是如此，这一行程序代码包含了以下三件事：

- 声明一个名为`products`且拥有一个`int`数组的数组变量。
- 创建一个拥有10个元素的数组，而每一个元素又是一个拥有10个`int`元素的数组。
- 创建10个数组，每一个数组都拥有10个`int`元素，然后再将这10个新创建的数组赋给最初声明的数组元素。这10个新的数组中的每一个`int`元素的默认值皆为0。

上述的程序代码也可以写成下面的样子：

```
int[][] products = new int[10][];    // 一个拥有10个int[]值的数组
for(int i = 0; i < 10; i++)          // 循环10次 ...
    products[i] = new int[10];       // ... 并创建10个数组
```

new 关键字会为你自动执行其他必需的初始化动作，另外它也可以用在多维数组中：

```
float[][][] globalTemperatureData = new float[360][180][100];
```

当多维数组使用new时，你不需要指定该数组所有维度的大小，只需要指定最左侧的几个维度就可以了。例如，以下的两行程序代码就是合法的：

```
float[][][] globalTemperatureData = new float[360][][];
float[][][] globalTemperatureData = new float[360][180][];
```

第一行会创建一个一维数组，该数组的每一个元素都拥有一个float[]。第二行会创建一个二维数组，每一个元素都是一个float[]。如果你只设定了数组中的某些维度，则这些维度必须位于最左侧才可以，因此以下的声明方式都是不合法的：

```
float[][][] globalTemperatureData = new float[360][][100]; // 错误！
float[][][] globalTemperatureData = new float[][180][100]; // 错误！
```

就和一维数组一样，多维数组也可以使用数组初始值来初始化，只需要用嵌套的大括号来表示嵌套数组的关系。例如，我们可以声明、创建并初始化一个如下的5×5的乘法表：

```
int[][] products = { {0, 0, 0, 0, 0},
                     {0, 1, 2, 3, 4},
                     {0, 2, 4, 6, 8},
                     {0, 3, 6, 9, 12},
                     {0, 4, 8, 12, 16} };
```

或者，如果你不想将多维数组声明为变量，你可以使用匿名初始化语法：

```
boolean response = bilingualQuestion(question, new String[][] {
    { "Yes", "No" },
    { "Oui", "Non" } });
```

当你使用new关键字来创建一个多维数组时，你所获得的一定是一个矩形数组：数组里每一个维度的大小都是相同的，这对矩形数据结构来说是相当适合的，如矩阵。因为在Java里多维数组是以“数组中的数组”的方式实现，而不是使用元素的单独矩形块来构造的，所以你没有被限制只能使用矩形数组。例如，乘法表是由左上到右下具对称性的，所以我们可以使用较少元素的非矩形数组来表示同样的信息：

```
int[][] products = { {0},
                     {0, 1},
                     {0, 2, 4},
                     {0, 3, 6, 9},
                     {0, 4, 8, 12, 16} };
```



当我们使用多维数组时，将会发现必须时常使用嵌套循环来创建或初始化它们。例如，你可以创建并初始化如下的乘法表：

```
int[][] products = new int[12][];           // 一个拥有 12 个 int 数组的数组
for(int row = 0; row < 12; row++) {         // 该数组中的每一个元素
    products[row] = new int[row+1];         // 分配一个 int 数组
    for(int col = 0; col < row+1; col++)     // 将每一个 int[] 的元素初始化
        products[row][col] = row * col;
}
```

## 引用类型

到目前为止我们已经讨论过数组、类与对象了，接下来我们要介绍一下引用类型。类与数组是 Java 的五种引用类型中的两种。类在稍早就已介绍过了，同时在第三章会和 interface 一起有更完整详细的说明。enumerated 类型与 annotation 类型是 Java 5.0 中才有的引用类型（请参阅第四章）。本章没有为这些特殊的引用类型做详细的语法说明，但却会说明引用类型的一般行为，并说明它们和 Java 的基本数据类型有何差异。同时在本节里，*object* 是用来指一个值或任何引用类型的实例，当然也包括了数组。

## 引用类型与基本数据类型

引用类型与基本数据类型在本质上有些不同：

- Java 语言定义了 8 个基本数据类型，而引用类型则是由用户自行定义的，所以它的数量是无限的。例如，有一个程序定义了一个名为 *Point* 的类而且使用了这个新定义类型的对象来存储与操作直角坐标系统的 X、Y 点。此程序会使用一个字符数组 *char[]* 来存储文本，同时也会使用一个 *Point* 对象数组 *Point[]* 来存储一连串的 *point*。
- 使用基本数据类型来表示一个单独的值。引用类型是拥有零个以上基本数据类型或对象的集合类型（aggregate type）。例如，我们所假设的 *Point* 类可以拥有两个 *double* 值来表示 X 与 Y 两点。*char[]* 与 *Point[]* 数组类型很明显就是集合类型，因为它们有一连串的基本字符值与 *Point* 对象。
- 基本数据类型需要 0 到 8 个字节的内存空间。当一个基本数据值存储于变量或传递给一个 *method* 时，计算机会将它们所拥有的值复制一份。*object* 就需要更多的内存了。*object* 存储于内存，当此 *object* 被创建时会在堆栈中动态地分配其位置，同时在此 *object* 不再需要使用时，它的内存就会自动地被回收（garbage-collected）。当 *object* 被指定给一个变量或被传递给一个 *method* 时，则该 *object* 就不需要被复制了，而是只引用存储该变量或该 *method* 的内存。

基本数据类型与引用类型的最后一点不同，是要说明为什么引用类型会被这样命名。接下来的章节会深入探讨这两种类型之间的差异点，也就是值与类型之间的运作及引用的运作。

考虑到引用的本质是需要的。引用基本来说就是一种对象的引用。在Java中，引用是完全不透明的，且引用的最佳实现就是Java解释器。如果你是C语言的程序员，可以将引用想象成是一个pointer或是内存地址，虽然Java程序中无法使用其他的方法来操作引用。与C与C++语言中的指针(pointer)不同的是，Java中的引用并不能被转换成整数或是由整数转换成引用，同时它们也不能够被递加或递减。C与C++的程序员也应该会注意到，Java并不支持&地址运算符或\*与->解引用运算符(dereference operator)。在Java中，处理基本数据类型的值，是把数值取出来操作，而处理object的值，则是通过引用的方式来操作：在Java中，运算符与C和C++的->运算符极为相似。

## 复制对象

下面的程序代码是用来操作基本int值的：

```
int x = 42;
int y = x;
```

在执行过这两行代码后，变量y会含有与变量x相同的值。在JVM中，则会有两个独立的32位整数值42。

现在思考一下将会发生什么事，如果我们运行同样的基本程序，但使用的是引用类型：

```
Point p = new Point(1.0, 2.0);
Point q = p;
```

在运行过这几行代码后，变量q会拥有与变量p相同的引用。在JVM中只会有一份Point对象的副本，但会有两份对该对象的引用的副本。这给了我们一些重要的暗示。假设上述的程序代码后面跟着以下几行程序代码：

```
System.out.println(p.x); // 输出p的x坐标: 1.0
q.x = 13.0;              // 现在改变q的x坐标
System.out.println(p.x); // 再一次输出p.x, 此次它的值为13.0
```

由于变量p与q拥有的引用指向相同的对象，所以这两个变量都可以被用来修改该对象，而修改过后的结果通过另外一个对象也可以看到。

像这样的行为不仅仅只限于对象，同样的事情也会发生在数组上，我们使用以下的程序代码来加以说明：

```
char[] greet = { 'h','e','l','l','o' }; // greet 拥有一个数组引用
char[] cuss = greet;                    // cuss 拥有和greet一样的引用
```

```
cuss[4] = '!';           // 使用引用来改变一个元素
System.out.println(greet); // 输出 "hell!"
```

基本数据类型与引用类型间的另一个不同点是发生在当自变量被传递给方法时,请看以下的代码:

```
void changePrimitive(int x) {
    while(x > 0)
        System.out.println(x--);
}
```

当此方法被调用时,它会获得一个自变量的副本以赋给参数 *x*。在方法中, *x* 是一个循环计数器且它会递减到零为止。因为 *x* 是一个基本数据类型,所以该方法自己拥有一个 *x* 值的副本,而这样的做法是可以理解的。

但另一方面,如果我们将该参数修改为引用类型时,会发生什么状况呢?

```
void changeReference(Point p) {
    while(p.x > 0)
        System.out.println(p.x--);
}
```

当此方法被调用时,一个指向 *Point* 对象的引用的私有副本会被传递给它并可以使用此引用来修改 *Point* 对象,请看以下的程序:

```
Point q = new Point(3.0, 4.5); // 一个 x 坐标为 3 的点
changeReference(q);           // 输出 3,2,1 并且修改 Point
System.out.println(q.x);      // 现在 q 的 x 坐标为 0 了!
```

当 *changeReference()* method 被调用时,它会获得一个与变量 *q* 所保存的相同的引用副本。现在变量 *q* 与 method 的参数 *p* 保存了指向相同的对象的引用,该 method 可以使用它的引用来修改该对象。然而,请注意,它并不能修改变量 *q* 里的内容。换句话说,该 method 会超过认可地修改 *Point* 对象,但它却不能改变 *q* 仍然引用同一个对象的事实。

本小节的标题是“复制对象”,但到目前为止我们只知道如何复制指向对象的引用,而非复制对象与数组本身。如果要实际地复制对象,你必须使用一个特殊的 *clone()* method (继承自 *java.lang.Object* 的所有对象都有此 method):

```
Point p = new Point(1,2); // p 指向一个对象
Point q = (Point) p.clone(); // q 指向该对象的副本
q.y = 42; // 修改对象副本,但不修改原始对象

int[] data = {1,2,3,4,5}; // 一个数组
int[] copy = (int[]) data.clone(); // 一个数组的副本
```

请注意你必须将 *clone()* method 的返回值强制转换为正确的类型。当你在使用 *clone()* method 时有几点是你必须特别注意的。第一,并不是所有的对象都可以被复



制, Java 只允许某些对象被复制, 而这些对象的类必须明确地通过 Cloneable 接口来声明它是可以被复制的 (我们尚未讨论到接口以及它们被实现的方式, 这些都会在第三章详加介绍)。我们先前所提到的 Point 定义并没有实际地实现这个接口, 所以我们的 Point 类型就不能被复制; 然而, 请注意, 数组一定是可被复制的。如果我们对一个不能被复制的对象调用 clone() method, 那么它会抛出 CloneNotSupportedException, 因此在你使用 clone() method 时, 最好将它放在一个 try 块中以捕获此异常。

第二件你必须了解有关 clone() 的事是在默认的状况下, 它只能为对象创建一个浅层 (shallow) 的复制。被复制的对象会包含原来对象中所有的基本或引用类型值。也就是说, 任何该对象的引用都会被复制, 但这些引用所引用的对象则不会被复制。类可能需要覆盖 (override) 这种浅层复制的行为, 而是自己定义它所需要的深层复制 (deeper copy) 的 clone()。为了要了解 clone() 的浅层复制行为, 请看以下的二维数组的复制:

```
int[][] data = {{1,2,3}, {4,5}};           // 一个拥有两个引用的数组
int[][] copy = (int[][]) data.clone();      // 复制这两个引用到一个新的数组
copy[0][0] = 99;                           // 改变了 data[0][0]
copy[1] = new int[] {7,8,9};               // 但并不会改变 data[1]
```

如果你想要对此多维数组执行深层复制, 则必须直接复制每一层:

```
int[][] data = {{1,2,3}, {4,5}};           // 一个拥有两个引用的数组
int[][] copy = new int[data.length][ ];    // 一个用来存放复制数组的新数组
for(int i = 0; i < data.length; i++)
    copy[i] = (int[]) data[i].clone();
```

## 比较对象

我们已经知道了基本数据类型与引用类型在赋值给一个变量、传递方法与复制时的运作方式有明显的不同, 而它们比较是否相等的方式也是不同的。基本数据类型的数值在比较是否相等时, 相等运算符 (==) 会测试两个数值是否一样 (也就是是否具有完全相同的位); 然而对引用类型而言, == 会比较引用, 而不是比较对象。换句话说, == 会测试两个引用是否访问相同的对象, 它并不会测试这两个对象的内容是否相同。例如:

```
String letter = "o";
String s = "hello";                          // 这两个 String 对象
String t = "hell" + letter;                  // 包含了同样的文字,
if (s == t) System.out.println("equal");    // 但它们并不相等 !

byte[] a = { 1, 2, 3 };                      // 一个数组
byte[] b = (byte[]) a.clone();               // 一个拥有同样内容的副本
if (a == b) System.out.println("equal");    // 但它们并不相等 !
```

在操作引用类型时, 有两个相等的情况: 引用相等与对象相等, 你必须将这两个相等区分得相当清楚。我们在这提供一个方法帮助你区分这两者: 当在提到引用相等时, 请使

用“相同”(identical)这个词；当在提到两个对象拥有同样的内容时，请使用“相等”(equal)这个词。要测试两个不同的对象是否相等时，可以把其中的一个对象传递给另外一个对象的 `equals()` method:

```
String letter = "o";  
String s = "hello";           // 这两个 String 对象拥有同样的文字内容  
String t = "hell" + letter;  
if (s.equals(t))              // equals() method 也是这样告诉我们的  
    System.out.println("equal");
```

所有的对象都会继承 `equals()` method (来自 `Object`)，但默认的实现只是使用 `==` 来测试引用相等，而不是内容相等与否。如果类想要能够比较对象的内容是否相同，则可以定义自己的 `equals()` method。我们的 `Point` 类并没有这样做，但 `String` 类有，正如上面程序代码所指示的。你可以调用数组的 `equals()` method，它的功能与 `==` 运算符相同，因为数组一定会继承自默认的 `equals()` method，它会比较引用相等，而不会比较数组的内容。你可以使用一个方便的 `java.util.Arrays.equals()` 来比较数组是否相同。

## 术语：通过值传递 (pass by value)

我们曾经提到过 Java 是“以引用”(by reference) 来操作对象，请不要将它与“以引用传递”(pass by reference) 相互混淆了。“pass by reference”只是一个用来描述某些程序语言的方法调用的惯用语法。在 pass-by-reference 的语言中，数值——即使是基本数据类型，都不会被直接传递给 method。method 只能获得对数值的引用，因此，如果 method 修改了它的参数，则当方法返回时，这些修改便会真正影响到这些数值，即使它们是基本数据类型的数值。

Java 并不是如此做的，它是一个“通过值传递”的语言，然而，当使用引用类型时，所传递的值是一个引用，但这又与 pass-by-reference 不同。如果 Java 是个 pass-by-reference 的语言，则当引用类型传递给 method 时，该 method 所获得的就会是一个对该引用的引用

## 内存分配与内存回收

我们先前已经提到过，对象是复合数据类型，它们可以拥有任意数量的值且需要相当的内存空间。当你使用 `new` 关键字创建一个新的对象或是在程序中使用一个对象或直接量时，Java 会自动地为你创建该对象，同时分配它所需要的内存空间。你不需要亲自去做这些事。

此外，Java 也会自动地回收已经不再使用的内存空间，它会通过内存回收 (garbage

collection) 机制来做这件事。当一个对象已经没有任何存储于变量、对象的字段、数组元素的引用指向它时, 它会被认为是没有用的。例如:

```
Point p = new Point(1,2);           // 创建一个对象
double d = p.distanceFromOrigin();  // 使用它来做某些事
p = new Point(2,3);                 // 创建一个新的对象
```

在Java解释器执行过第三行代码之后, 该引用会指向一个新的Point对象, 而不再指向旧的Point对象。现在已经没有任何的引用指向该旧的对象, 因此它变成了一个无用的对象。内存回收程序 (garbage collector) 将会发现这一点, 并将该对象所使用的内存给释放出来。

C程序员习惯使用malloc()与free()来管理内存, 而C++程序员则习惯使用delete来删除对象, 似乎认为将这些事情都由内存回收程序来处理是一件不太令人放心的事。虽然它看起来有点不可思议, 但它真的就是能百分之百达到这样的功能。当使用内存回收程序时, 性能会有些微的下降, 而Java程序在运行中速度有时候也会突然地变慢, 这是因为内存回收程序正在回收内存的关系。然而, 将内存回收的功能内置于语言中会降低内存泄漏 (memory leak) 的发生几率, 程序错误数也会相对地减少, 并改善程序员的生产效率。

## 引用类型的转换

对象可以在不同的引用类型之间相互地转换。和基本数据类型一样, 引用类型的转换也有放大转换 (widening conversions, 编译器会自动地执行) 和缩小转换 (narrowing conversions, 需要使用强制转换 (cast) 明确地指示它必须如此做)。为了了解引用类型的转换机制, 你必须以层次的观点来了解该类型, 通常我们称这为类层次 (class hierarchy)。

每个Java类都扩展了 (extend) 某个其他的类, 也就是父类 (superclass)。每个类都继承了它的父类的字段与方法, 并定义了自己的字段与方法。有一个名为Object的特殊类, 它是Java中类层次的根 (root), 所有的Java类都是直接或间接地扩展自Object。Object类定义了许多特殊的方法来让所有类继承 (或加以覆盖)。

先前我们所定义的String类与Point类都是扩展自Object, 因此, 我们可以说所有的String对象都是Object对象, 所有的Point对象也都是Object对象。但你却不可以反过来说所有的Object对象都是String对象, 因为有些Object对象是Point对象。

在初步了解类层次之后, 我们可以回来讨论引用类型转换的规则:



- 对象不能被转换为无关的类型。Java编译器并不允许你将String转换为Point,即使你使用了强制转换运算符也不行。
- 对象可被转换为父类的类型,这是一个放大转换,所以不需要强制转换。例如,一个String值可被赋给一个类型为Object的变量,或是传递给某个需要一个Object类型的参数的method。请注意,实际上并没有任何的转换动作被执行,该对象只是被当作是父类的一个实例而已。
- 对象可以被转换为子类的类型,但这是缩小转换,因此需要强制转换。Java编译器会在运行时检查其是否合法。你只能在你所确定的对象上使用强制转换来将它转换为某个子类的类型,逻辑上,该对象会成为该子类的一个实例。如果此转换不合法,解释器就会抛出ClassCastException。例如,如果我们将一个Object对象赋值给一个类型为String的变量时,必须使用强制转换:

```
Object o = "string";    // 从String到Object的放大转换
// 程序代码 ...
String s = (String) o;  // 从Object到String的缩小转换
```

所有的数组都算是对象的实例且它会依循着自己的转换规则。首先,任何对象通过放大转换都可以被转换为Object值,使用强制转换的缩小转换也可以将对象的值转换回数组类型的值。例如:

```
Object o = new int[] {1,2,3}; // 从数组到Object的放大转换
// 程序代码 ...
int[] a = (int[]) o;          // 从Object到数组类型的缩小转换
```

除了将数组转换为Object外,如果两个数组的类型是可以互相转换的类型,则数组也可以被转换为其他类型的数组。例如:

```
// 这是一个strings的数组
String[] strings = new String[] { "hi", "there" };
// 将其放大转换为CharSequence[]是被允许的,因为String可以被放大转换为CharSequence
CharSequence[] sequences = strings;
// 缩小转换回String[]需要使用强制转换
strings = (String[]) sequences;
// 这是strings数组的数组
String[][] s = new String[][] { strings };
// 它不可以被转换为CharSequence[],因为String[]不能够被转换为CharSequence:
// 维度的数目不相符
sequences = s; // 此行将无法编译
// s可以被转换为Object或Object[],因为所有的数组类型(包括了String[]
// 与String[][])都可以被转换为Object
Object[] objects = s;
```

请注意,这些数组的转换规则只适用于数组对象与数组组成的数组。一个基本数据类型的数组不能够被转换为任何其他的数组类型,即使它们之间的元素是可以转换的。

```
// 无法将 int[] 转换为 double[], 即使 int 可以被放大转换为 double
double[] data = new int[] {1,2,3}; // 此行会造成编译上的错误
// 此行是合法的, 因为 int[] 可以被转换为 Object
Object[] objects = new int[][] {{1,2},{3,4}};
```

## Boxing 和 Unboxing 转换

基本数据类型与引用类型的运作方式是完全不同的。有时候将基本数据类型的值视为对象是很有用的, 就是因为这样, Java 平台为每一个基本数据类型增加了 wrapper class。Boolean、Byte、Short、Character、Integer、Long、Float 与 Double 是固定 class, 这些 class 的实例都拥有与其相对应的基本数据类型值。当你想要将这些基本数据类型的值存储于 collection 时, 都会使用到这些 wrapper 类, 例如 java.util.List:

```
List numbers = new ArrayList(); // 创建一个新的集合
numbers.add(new Integer(-1)); // 存储一个 wrapped 数据
int i = ((Integer)numbers.get(0)).intValue(); // 取得基本数据值
```

Java 5.0 以前的版本并没有提供基本数据类型与引用类型之间直接转换的功能。此程序代码明确地调用 Integer() 构造函数将 int 值封装成一个对象, 同时它也调用了 intValue() method 于 wrapper 对象中取出基本数据类型值。

Java 5.0 新增了两个转换类型, 就是我们所知道的 boxing 与 unboxing 转换。boxing 转换会将基本数据类型的值转换为与它相对应的 wrapper 对象, 而 unboxing 转换则是做相反的操作。虽然可以使用强制转换 (cast) 来表示 boxing 或 unboxing 转换, 但你不需要这样做, 因为这些转换在你赋一个值给变量或传递一个值给 method 时, 都会被自动执行。此外, 当 Java 运算符或语句是基本数据类型时, 如果你使用了它的 wrapper 对象, 则 unboxing 转换也是会自动执行的。Java 5.0 提供了自动 boxing 与 unboxing 的功能, 这个新的功能就是 autoboxing。

这里有一些关于自动 boxing 与 unboxing 转换的例子:

```
Integer i = 0; // int 直接量 0 被封装 (boxed) 成一个 Integer 对象
Number n = 0.0f; // float 直接量被封装成 Float 且被放大为 Number
Integer i = 1; // 这是一个 boxing 转换
int j = i; // i 在这里被解封装 (unboxed)
i++; // i 被解封装, 递加, 然后又被封装一次
Integer k = i+2; // i 被解封装, 而且总和又被封装一次
i = null;
j = i; // 在这里解封装, 且抛出一个 NullPointerException 异常
```

自动的 boxing 与 unboxing 转换使得基本数据值使用 collection 类变得更加容易。在本章节前面的一些程序代码都可以转变为 Java 5.0 所提供的这种用法, 同时这样的转换也可以使用在泛型 (generic) 上。Java 5.0 的其他新功能将会在第四章中做说明。

```
List<Integer> numbers = new ArrayList<Integer>(); // 创建一个 Integer 类型的 List
numbers.add(-1); // 将 int 封装成 Integer
int i = numbers.get(0); // 将 Integer 解封装成 int
```

## 包与 Java 命名空间

包是由一组类、接口与引用类型所组成的。包将类聚集在一起，并为它们所包含的类定义了命名空间（namespace）。

Java 平台的核心类都在名称以 Java 开头的包中。例如，Java 语言中最基础的类是位于包 `java.lang` 中，而各种工具类则是位于 `java.util` 中，处理输入与输出的类位于 `java.io` 中，用于网络的类则位于 `java.net` 中。这些包中有些还包含分包（subpackage），例如 `java.lang.reflect` 与 `java.util.regex`。Sun 也定义了 Java 平台的标准扩充版，它的包名以 `javax` 开头。这些扩充版，如 `javax.swing` 与各式各样的分包，都被用在后来的 Java 平台内。最后，Java 平台也包括了几个“被认可的标准（endorsed standards）”，如 `org.w3c` 与 `org.omg`。

每一个类都拥有一个简称（simple name）与一个完全限定名称（fully qualified name）。简称是在它的定义中所指定给该类的名称，而完全限定名称则包含了该类所在包的名称。例如，`String` 类是位于 `java.lang` 包里，因此它的完全限定名称为 `java.lang.String`。

本章节将说明如何将你自己的类与接口放入包中，以及如何选择一个包名而它不会与其他包名有所冲突。再次会说明如何选择性地将输入的类型名称放入命名空间中，使你不需输入所使用的类或接口的包名。最后，将会说明 Java 5.0 中的新功能：将静态成员导入命名空间中，使你不需要将包名或类名放在前面。

## 包的声明

为了将类指定给包，你必须使用 `package` 声明。`package` 关键字如果出现的话，它必须是 Java 文件中 Java 程序代码里的第一个符号。此关键字之后必须紧接着该包的名称与一个分号。以下为以它为起始的 Java 程序代码：

```
package com.davidflanagan.examples;
```

该文件所定义的所有类都是位于名为 `com.davidflanagan.examples` 的包中。

如果在 Java 程序代码中并没有使用 `package` 指令，则所有在该文件里所定义的类都是属于一个默认且未命名的包。为了防止出现命名冲突，你应该只在简单的程序代码或开发大型项目的初期使用默认包。



## 全局唯一包名

包有一个相当重要的功能，就是分配 Java 的命名空间，以避免类间发生名称冲突。`java.util.List` 与 `java.awt.List` 两个类的包名不同，也使得类之间不会因名称相同而相互冲突。包的名称是不能相同的。身为 Java 的开发者，Sun 掌控了所有以 `java`、`javax` 与 `sun` 为开头的包名。

对我们来说，Sun 提供了一个包命名规则，如果遵守的话，将可以确保包拥有全局唯一包名。此规则是使用你的 Internet 域名，将其中的组成元素的次序颠倒过来，作为你的包名的开头部分。我的网站地址是 `http://devidflanagan.com`，因此我所有的包都会以 `com.davidflanagan` 开头。要如何分割 `com.davidflanagan` 之下的命名空间由我自己来决定，正因为我拥有此域名，所以没有其他的人或组织会因此命名规则而发生与我的包名相同的情形。

请注意，这些包命名规则也适用于 API 的开发者。如果其他的程序员要使用你自行开发的类时，则你的包名的全局唯一性就变得非常重要了。另一方面，如果你正在开发一个 Java 应用程序，而且还无法将某些类释放出来给其他的程序使用时，你会知道自已的应用程序里有哪些完整的类将要被部署，而不须要担心会发生名称的冲突。当然，你也可以选择你自己认为方便的包命名规则。一个常见的方法是使用应用程序的名称来当作主要包的名称（它不适合于分包）。

## 导入类型

当在 Java 程序代码里引用类或接口时，默认情况下，你必须使用它的完全限定名称，其中包含包的名称。如果程序代码是用于操作一个文件，同时需要使用 `java.io` 包内的 `File` 类时，则必须输入 `java.io.File`。这个规则有三个异常：

- `java.lang` 包内的类型是重要且常常都会被使用到，所以只需通过它们的简称来引用就可以了。
- `p.T` 类型的程序代码可以通过其简称来引用定义于包 `p` 中的其他类型。
- 已用 `import` 声明被导入命名空间中的类型，可以通过它的简称来引用。

前两个异常是所谓的“自动导入”。`java.lang` 的类型与常用的包已经“导入（import）”到命名空间，所以在使用时不需要它们的包名。不断地导入包名是一件相当累人的事，所以明确地从其他包导入类型也是可能的。这可以用 `import` 声明实现。

`import` 声明必须出现在 Java 文件的开始处，它是紧接在 `package` 声明之后出现，并且在任一个类型定义之前。在文件里你可以使用多个 `import` 声明。`import` 声明适用于文件中的所有类型定义。

import 声明有两种形式，一种是将单独的类型导入命名空间中，紧接在 import 关键字之后的是该类型的名称与分号：

```
import java.io.File;    // 现在我们可以输入 File，而不是 java.io.File
```

这就是“单独类型导入 (single type import)”声明。

另一种形式是“随选类型导入 (on-demand type import)”。在这样的形式中，你可以在包名之后输入 `*` 字符来指出该包内可能会用到的其他类型，而不需要使用包名。因此，如果除了 `File` 类之外，你还要使用 `java.io` 包中的其他类时，你可以导入整个类：

```
import java.io.*;    // 现在我们可以使用 java.io 里的所有类的简称了
```

这种随选 import 的语法并不适用于分包，如果我导入了 `java.util` 包，还是必须使用完全限定名称来使用 `java.util.zip.ZipInputStream` 包。

对于包中的每个类来说，使用随选类型导入声明和明确地写出单独类型导入声明并不相同。对于你在程序代码中实际使用到的包中的每个类型，就很像是明确的单独类型导入。这也就是它被称为“随选”的原因：类型会在你使用到时被导入。

## 命名冲突与复制

import 声明对 Java 程序来说是非常重要的。但是，它会让我们遇到命名冲突。考虑 `java.util` 与 `java.awt` 这两个包，它们都有一个名为 `List` 的类。`java.util.List` 是个重要且经常使用的接口。`java.awt` 包包含了一些重要的类型，它们常被使用于客户端 (client-side) 的应用程序，但 `java.awt.List` 已经被取代了，而且它也不是这些重要类的其中一个。在 Java 文件里同时将 `java.util.List` 与 `java.awt.List` 导入是不合法的。以下的单独类型导入声明会产生编译错误：

```
import java.util.List;
import java.awt.List;
```

对这两个包使用随选类型导入是合法的：

```
import java.util.*;    // 针对 collection 与其他公用程序
import java.awt.*;     // 针对字型、色彩与图形
```

但是，如果你真的试着要使用 `List` 类型时就会遇到困难。这个类型会从这两个包的其中一个“随选”，而且任何试图使用不合法类型名称的 `List` 都将会产生编译错误。在这样的状况下，解决方法就是明确地指定你要的包名。

因为 `java.util.List` 比 `java.awt.List` 还常被使用到，所以将两个随选类型导入声明与一个单独类型导入结合起来以明确我们所指的 `List` 是什么，是很有用的：

```
import java.util.*;    // 针对 collection 与其他公用程序
import java.awt.*;     // 针对字型、色彩与图形
import java.util.List; // 与 java.awt.List 分清
```

适当地使用 import 声明来指定包，我们就可以使用 List 来表示 java.util.List 接口。如果我们实际上需要使用 java.awt.List 类的话，只要加上它的包名，就仍做得得到。java.util 与 java.awt 这两个包之间没有其他的命名冲突，当我们在没有加上包名的情况下使用它们的类型时，就会“随选”导入。

## 导入静态成员

在 Java 5.0 及之后的版本，你都可以使用 import static 关键字来导入类型的静态成员 (static member) 及其本身 (静态成员会在第三章中说明。如果你对它们不熟悉，或许稍后要再看这节)。就像类型导入声明一样，这些 static import 声明有两种形式：单独静态成员导入 (single static member import) 与随选静态成员导入 (on-demand static member import)。例如，假设你正在写一个以文本为基础的程序，它会将很多的输出送到 System.out。在这样的状况下，你可以用单独静态成员导入来减少导入。

```
import static java.lang.System.out;
```

在加上此导入声明后，你就可以用 out.print() 来取代 System.out.print()。或者，假设你正在写一个程序，该程序使用了很多三角函数与 Math 类中的其他函数。在程序中可以很明显地看到都是集中在处理数值的 method 上，不断地重复输入“Math”这个类名并不会让你的程序代码更清楚，只是必须这么做。在这样的状况下，使用随选静态成员导入或许更适合：

```
import static java.lang.Math.*
```

在这样的导入声明后，你就可以写出很简单的表达式，例如 sqrt(abs(sin(x))) 这样，而不需要在每个静态 method 的名称前面加上“Math”类名。

另一个 import static 声明的重要用法是将常量的名称导入程序代码中。这特别适合枚举类型 (enumerated type，请参阅第四章)。例如，假设你想要在程序里使用枚举类型的值，可以写成：

```
package climate.temperate;
enum Seasons { WINTER, SPRING, SUMMER, AUTUMN };
```

你可以输入 climate.temperate.Seasons 这个类型，然后在常量前面加上类型名称：Seasons.SPRINT。如果要写得更简单点，你可以导入枚举值本身：

```
import static climate.temperate.Seasons.*;
```

针对常量使用静态成员导入声明与实现定义常量的接口相比，前者通常是较佳的做法。



## 静态成员导入与重载 method

静态导入声明导入了名称，而不是具有该名称的任意一个特定成员。因为 Java 允许 method 的重载，并允许一个类型可以有相同名称的字段与 method，所以单独静态成员导入声明实际上可能会导入多个成员。考虑这段程序代码：

```
import static java.util.Arrays.sort;
```

这个声明会将名称“sort”导入命名空间，而该“sort”并不是由 java.util.Arrays 定义的 19 个 sort() method 里的任何一个。如果你使用了导入的名称 sort 来调用 method，编译器就会去查看该 method 自变量的类型，以决定你指的是哪个 method。

只要各 method 拥有不同的签名，则从两个或多个不同的类型中导入具有相同名称的静态 method 都是合法的。这里有个范例：

```
import static java.util.Arrays.sort;
import static java.util.Collections.sort;
```

你可能预期这样的程序会发生语法上的错误，但事实上并没有，因为 Collections 类所定义的 sort() method 与 Arrays 类所定义的 sort() method 各自拥有不同的签名。当你在程序中使用了“sort”这个名称，编译器就会去查看自变量的类型来决定在 21 个可能的已导入的 method 里，你指的是哪一个。

## Java 文件结构

本章已经带领我们从 Java 语法最小的元素到最大的元素，从个别的字符与符号到运算符、表达式、语句与 method，最后介绍了类与包。就实用的观点来看，你最常用到的 Java 程序结构单元是 Java 文件。Java 文件是可被 Java 编译器所编译的最小单元。一个 Java 文件包括了：

- 一个可选的 package 指令
- 零个或多个 import 或 import static 指令
- 一个或多个类定义

当然，这些元素之间都可以加上注释，但必须注意的是，这三个元素必须依此顺序。对一个 Java 文件来说，这就是全部了。所有的 Java 语句（除了 package 与 import 指令外，它们不是真正的语句）都必须在 method 里出现，而所有的 method 都必须出现在类型定义（type definition）中。

Java 文件还有另外两个很重要的规定。第一，每一个文件最多只能有一个被声明为

public 的类。public 类是被设计成可以被其他包中的其他类来使用的。在公共类上的这个规定只适用于最上层 (top-level) 的类；一个类可以含有任意数量的被声明为 public 的嵌套类或内部类 (inner class)。我们将在第三章看到更多关于 public 修饰符与嵌套类的内容。

第二个规定是关于 Java 文件的名称。如果 Java 文件包含了一个 public 类，则该文件的名称必须和该类的名称相同，并在后面加上 .java。因此，如果 Point 是一个被定义为 public 的类，则它的程序代码必须在名称为 Point.java 的文件内。不管你的类是不是 public，一个文件只定义一个类且该文件的名称与类的名称相同是一个好的习惯。

当一个 Java 文件被编译时，它所定义的每一个类都会被编译到一个独立的类文件中，该文件包含了 Java 字节码而且可以被 JVM 解释。一个类文件的名称必须和它所拥有的类的名称相同，只是在后面加上 .class 而已。因此，如果 Point.java 定义了一个名为 Point 的类，则 Java 编译器会将它编译到一个名为 Point.class 的文件中。在大部分的系统中，类文件会依它们的包名存储于目录中。因此，类 com.davidflanigan.examples.Point 是由 com/davidflanigan/examples/Point.class 所定义的。

Java 解释器知道标准系统类的类文件要从何处加载，并且在需要的时候加载必要的文件。当解释器在运行一个需要使用名称为 com.davidflanigan.examples.Point 的类的程序时，它知道该类的程序代码是存储于名称为 com/davidflanigan/examples 的目录内，并且在默认情况下，它会“查看”当前的目录里是否拥有该名称的子目录。如果要解释器去其他的地方搜索，则在调用解释器时，就必须使用 -classpath 参数或是设定 CLASSPATH 环境变量。在第八章将会更详细地介绍 Java 解释器：java。

## 定义并运行 Java 程序

Java 程序包含了一组相互作用的类定义，但并非所有的 Java 类或 Java 文件都定义了一个程序。为了创建一个程序，你必须定义一个具有以下特征的特殊方法的类：

```
public static void main(String[] args)
```

main() method 是程序的主要进入点，Java 解释器会从这里开始运行。此 method 会获得一个字符串数组，但并没有返回值。当 main() 完成运行时，Java 解释器便会跳出，同时也结束运行（除非 main() 另外创建其他线程，在这样的情况下，解释器会等到所有的线程都运行完毕才跳出）。

如果要运行 Java 程序，就要运行 Java 解释器，java，并指定含有 main() method 的类的完全限定名称。请注意，你所指定的是类的名称，而不是含有此类的文件的名称。你

在命令行上所指定的其他自变量都会被传递给 `main()` method，成为它的 `String[]` 参数。你也许需要设定 `-classpath` 参数来告诉解释器去寻找该程序所需要的类。考虑以下命令：

```
% java -classpath /usr/local/Jude com.davidflanagan.jude.Jude datafile.jude
```

`java` 是用来运行 Java 解释器的命令，`-classpath /usr/local/Jude` 则告诉解释器到哪儿去寻找 `.class` 文件。`com.davidflanagan.jude.Jude` 是该程序所要运行的文件的名称（例如定义了 `main()` method 的类的名称）。最后，`datafile.jude` 是要传递给 `main()` method 的自变量。

还有一个较简单的运行程序的方法。如果程序及其辅助类（除了 Java 平台中的类）都已经被正确地绑定在一个 Java 压缩文档（JAR）中，你便可以直接指明 JAR 文件的名称来运行程序：

```
% java -jar /usr/local/Jude/jude.jar datafile.jude
```

有些操作系统会让 JAR 文件成为自动执行文件。在这些系统中，你只要输入：

```
% /usr/local/Jude/jude.jar datafile.jude
```

在第八章里将会更详细地讨论。

## Java 与 C 的不同点

如果你是 C 或 C++ 的程序员，你应该已经发现 Java 的许多语法规则（尤其是运算符与语句）与 C 或 C++ 十分的类似。因为 Java 与 C 在某些方面非常的相似，而对 C 与 C++ 程序员来说，了解它们之间的不同点便显得格外重要了。以下就是我们所整理出来的关于 Java 与 C 的不同之处：

### 没有预处理器

Java 没有预处理器（preprocessor），因此并未定义类似 `#define`、`#include` 与 `#ifdef` 的指令。Java 中的常量定义也被 `static final` 字段所取代（例如 `java.lang.Math.PI` 字段）。在 Java 里也可定义宏，但高级的编译技术和内联（inlining）已让宏较无用处。Java 并不需要 `#include` 指令，这是因为 Java 并没有头文件（header file）。Java 类文件同时含有类 API 与该类的实现，而编译器会在需要时从类文件中读取 API 信息。Java 也没有任何形式的条件编译模式，但因为它具有跨平台的可移植性，所以代表此特性并不十分需要。



### 没有全局变量

Java定义了一个非常整洁的命名空间。包包含了类，类包含了字段与方法，而方法则包含了局部变量。但Java中并没有全局变量，所以在变量中不可能发生名称冲突的状况。

### 明确定义的基本数据类型大小

Java中的所有基本数据类型都具有定义明确的大小，但在C中，short、int与long类型的大小则会依据平台而不同，这使得C的可移植性受到了阻碍。

### 没有指针

Java类与数组都是引用类型，对对象与数组的引用和C中的指针相当类似。然而，与C的指针不同的是，Java中的引用相当复杂，你没有办法将一个引用类型转换为基本数据类型，而且引用类型也无法递加或递减。没有像&的地址运算符，也没有像\*或->的解引用运算符，也没有sizeof运算符。许多程序错误的产生原因都是因为指针。将它们消除可简化语言，并且让Java程序更强壮、更安全。

### 内存回收

JVM会执行内存回收的操作，因此，Java程序员不需为内存管理方面的问题而烦恼。这个特点也大大地减少了程序错误的发生机会，同时也降低了Java程序发生内存泄漏的几率。

### 没有goto语句

Java并不支持goto语句，除了一些特定情况之外，在程序中使用goto语句通常被认为是相当糟糕的实践方法。Java除了拥有C所提供的流控制语句之外，还加入了异常处理以及break和continue语句，它们都是goto的极佳替代品。

### 可在任何地方声明变量

C必须在method或程序块的最开头声明局部变量，Java则可以在method或程序块的任何地方声明。然而，许多程序员还是比较喜欢将所有的变量集中起来，放在method的最前头。

### 向前引用 (forward reference)

Java编译器比C编译器聪明。在Java中，method可在定义之前被调用。你不需像在写C程序那样，在在程序文件中定义它们之前，就先在头文件中声明函数。

### 方法重载 (method overloading)

Java程序可将多个method定义为相同的名称，只要它们的参数列表不同就可以了。

### 没有struct与union类型

Java不支持C程序里的struct与union类型。Java类可被想象成是加强版的Struct。

### 没有位域 (bitfield)

Java 不支持 C 程序的另一项功能 (很少会用到的一项功能), 即设定 `struct` 中的字段所占的单独位数。

### 没有 typedef

Java 不支持 C 程序的 `typedef` 关键字, 它是用来定义类型名称的别名。由于 Java 没有指针, 所以它的类型命名规则比 C 的更简单、一致, 而且, `typedef` 的许多功能在 Java 中其实也用不到。

### 没有方法指针

C 程序允许你将函数的地址存储于变量中, 并将此函数的指针传递给其他的函数。但对 Java `method` 来说, 你不能这样做, 但你可以通过传递一个用来实现某个特定接口的对象来达到类似的效果。同时, `Java method` 也可经由 `java.lang.reflect.Method` 对象来表示与调用。





# Java 的面向对象 程序设计

在介绍过 Java 语言的基本语法后，现在我们开始介绍 Java 的面向对象程序设计。所有的 Java 程序都是使用对象以及由类或接口所定义的对象类型的方式写成的。每一个 Java 程序都被定义为一个类，而比较复杂一点的程序通常都会加入一些类与接口的定义。本章将会介绍如何定义一个新的类与接口以及如何使用它们来做面向对象程序设计。

本章相当的长同时也相当的详细，所以我们要从面向对象的概念与定义开始。类是由字段集合而成的，且该字段拥有了值与运作于这些值上的 method。类是所有 Java 程序中最基础的元素，写程序的时候你不可以没有定义类。所有的 Java 语句都出现在 method 里面，而且所有的 method 都是在类里面（注 1）。

类可以定义一个新的引用类型，如第二章所定义的 Point 类型。对象是类的实例。Point 类定义了一个类型，该类型是二维空间内所有可能的点的集合，而 Point 对象是该类型的值，它代表了一个单独的二维空间内的点。对象的创建常常是使用 new 关键字与调用一个构造函数来实例化一个类，就如这里所显示的一样：

```
Point p = new Point(1.0, 2.0);
```

注 1：如果你没有面向对象（OO）程序设计的背景，请不要担心，本章并没有假设你以前有任何的经验。然而，如果你有 OO 的程序设计经验，那么也请仔细阅读，在不同的程序语言里“面向对象”这个专有名词会有不同的意义。不要以为 Java 会和你所爱用的 OO 语言有着相同的运作方式，尤其是 C++ 程序员。虽然 Java 与 C++ 采用了 C 语言中相当多的语法，但这两种语言之间的相似度在语法上并非没有太大的差异，就因为这样，所以请有 C++ 经验的程序员在学习 Java 时要特别的小心。



类定义是由签名 (signature) 与主体 (body) 所组成的。类的签名定义了该类的名称, 同时也说明了其他重要的信息。而类的主体则是在大括号里头的一组成员 (member), 这些成员包含了字段与 method、构造函数与初始化程序以及嵌套类型。

成员可以是静态或非静态的。静态成员 (static member) 是属于类自己的, 而非静态成员 (nonstatic member) 则是与该类的实例相关联的 (请看本章的“字段与 method”一节)。

类的签名可以声明该类是扩展自其他的类, 被扩展的类就是我们所称的父类, 而该扩展的部分就是我们所称的子类。一个子类继承了它的父类的成员, 同时它也可以声明一个新的成员或重写 (override) 继承而来的 method。

类的签名也可以声明该类实现的一个或多个接口。接口 (interface) 是一个引用类型, 且该引用类型定义了 method 签名, 但不包含实现 method 的 method 主体。类在实现接口时为该接口的 method 提供主体是必须的。这样的类实例也是它所实现的接口类型的实例。

类的成员也可以有 public、protected 或 private 这样的访问修饰符, 这些修饰符说明了该类以及其子类的透明度与可访问程度, 同时它也允许类将不是公共 API 的部分成员给隐藏起来。当用于字段时, 这种将成员隐藏的能力就是面向对象设计里的数据封装 (data encapsulation) 技术。

类与接口是 Java 所定义五个基本引用类型中最重要的两个类型, 而 array、enumerated (或是“enums”) 与 annotation 类型则是其他三个引用类型。数组已经在第二章介绍过了, enumerated 与 annotation 类型是 Java 5.0 中才加入的类型 (请看第四章), enums 是一种很特殊的类, 而 annotation 类型则是一种很特殊的接口。

## 类的定义语法

类的定义是由关键字 class 与接在它后面的类的名称以及大括号内的一组类成员所组合而成的。修饰符关键字与 annotation 会在 class 关键字的前面 (请看第四章)。如果类扩展自其他的类, 则在类名之后要加上 extends 关键字与被扩展的类的名称。如果该类实现了一个或多个接口时, 则该类名或 extends 子句之后就需加上 implements 关键字并使用逗号来分隔接口名称列表。例如:

```
public class Integer extends Number implements Serializable, Comparable {  
    // 类成员从这里开始  
}
```

泛型类 (generic class) 的声明包括了一些附加的语法, 在第四章中会做更深入的介绍。

类的声明可以包含零个或多个下面所述的修饰符：

`public`

`public` 类可被该包之外所定义的类看见。请参阅稍后的“数据的隐藏与封装”一节。

`abstract`

`abstract` 类的实现是不完整的，而且它无法被实例化。任何的类，若它有一个或多个 `abstract method` 时，则该类必须被声明为 `abstract`。

`final`

当类使用 `final` 修饰符来声明时，意味着不能用该类来定义子类。声明为 `final` 的类，在调用该类的 `method` 时，它可以让 JVM 将该 `method` 最佳化。

`strictfp`

如果有一个定义为 `strictfp` 的类，则它的所有 `method` 的行为就会像它们所声明的 `strictfp` 那样。这是个很少被使用到的修饰符，在第二章的“Method”一节里就已介绍过了。

类不能同时为 `abstract` 与 `final`。习惯上，如果类有超过一个以上的修饰符时，则它们的出现是会有顺序的。

## 字段与 method

类可被看成数据与运作在数据上的程序代码的集合。数据被存储于字段里，而程序代码则是有组织地被放入 `method` 中。本小节就是介绍字段与 `method` 这两个最重要的类成员（`class member`）。字段与 `method` 有两个不同的类型：类成员（也是所谓的 `static member`）与该类本身是相关联的，而实例成员（`instance member`）则是与该类的某个实例（即对象）相关联。我们有四种成员：

- 类字段
- 类 `method`
- 实例字段
- 实例 `method`

例 3-1 是一个名为 `Circle` 的类的简单定义，其中包含了这四种成员。

例 3-1：一个简单的类与其成员

```
public class Circle {  
    // 一个类字段
```

```
public static final double PI= 3.14159;    // 一个有用的常量
// 一个类 method: 使用自变量来计算数值
public static double radiansToDegrees(double rads) {
    return rads * 180 / PI;
}

// 一个实例字段
public double r;                          // 圆的半径

// 两个实例 method: 它们以对象的实例字段来加以运算
public double area() {                    // 计算圆的面积
    return PI * r * r;
}
public double circumference() {          // 计算圆的周长
    return 2 * PI * r;
}
}
```

以下的章节将会依次介绍这四种成员。首先,我们要先说明的是字段声明语法 (method 声明语法在稍后的“method”一节中介绍)。

## 字段声明语法

字段声明的语法与声明局部变量的语法非常的相似 (请看第二章), 而字段的定义也可以包含修饰符。最简单的字段声明是在字段类型之后紧跟字段名称, 而在该字段类型之前可以有零个或多个修饰符或 annotation (请参阅第四章), 同时在该字段名称之后可以加入等号与提供字段初始值的初始化表达式。如果有两个或多个字段共享相同的类型与修饰符时, 则在该字段类型之后使用逗号将一连串字段名称与初始值分隔开。以下是一些合法的字段声明:

```
int x = 1;
private String name;
public static final DAYS_PER_WEEK = 7;
String[] daynames = new String[DAYS_PER_WEEK];
private int a = 17, b = 37, c = 53;
```

字段的修饰符是由零个或多个以下所述的修饰符组合而成的:

public、protected、private

这几个访问修饰符是说明该字段是否可被类之外的其他类所使用。这些重要的修饰符在稍后的“数据隐藏与封装”一节会做说明。这些访问修饰符可以出现在字段声明里面。

static

此修饰符是说明该字段与定义类本身相关联而不是与类的实例相关联。



### final

使用 `final` 修饰符来声明时,就意味着一旦该字段被初始化后,则该字段的值就不允许被改变。字段若同时拥有了 `static` 与 `final` 这两个修饰符,则表示它是编译期的常量,编译器会将它给内联化 (`inline`)。 `final` 字段也可以用于创建一个永远都不会变的实例。

### transient

此修饰符是说明该字段不是对象的持续状态的一部分,而当它与其他对象在一起时,将不被序列化 (`serialized`)。在第五章会对序列化 (`serialization`) 做详细的说明。

### volatile

粗略地说, `volatile` 字段就像是 `synchronized method`: 两个或两个以上的线程同时使用一个资源的情况是安全的。更精确地说, `volatile` 是表示该字段的值必须一直从主存储器中被读取与释放,且它无法被线程高速缓存 (在缓存器或CPU中高速缓存)。

## 类字段

类字段是与定义它的类相关的,而不是与类的实例相关联的。以下便是声明一个类字段的程序代码:

```
public static final double PI = 3.14159;
```

上述的代码声明了一个 `double` 类型且名为 `PI` 的字段,同时将它的值指定为 `3.14159`。正如你所看到的,字段的声明看起来与局部变量的声明十分的相似。当然,也有不同之处:变量只能在 `method` 中被定义,而字段则是类成员。

`static` 修饰符表示该字段是一个类字段。类字段有时候被称作静态字段,就是因为 `static` 修饰符的关系。`final` 修饰符表示该字段的值是不能被变动的,因为字段 `PI` 代表一个常量,所以我们将它声明为 `final` 让它不会被修改。这是 Java 中 (在许多其他程序语言中也可见到的) 的一个惯例,也就是常量通常都会使用大写来表示,这也就是为什么我们的字段名称为 `PI`,而不为 `pi` 的原因。定义这样的常量经常会使用类字段,也就是说 `static` 与 `final` 修饰符会常常被放在一起使用。然而,并不是所有的类字段都是常量。换句话说,字段可以被声明为 `static` 而不被声明为 `final`。最后, `public` 修饰符表示任何人都可以使用该字段,这是一个常见的修饰符,而且我们会在稍后的章节更详细地介绍它与其他的相关修饰符。

了解静态字段 (`static field`) 的一个重点是,对静态字段而言,永远只存在一份副本。此字段是与该类本身相关,而不是与该类的实例相关。如果你看到了 `Circle` 类中的许多

method, 你将会发现它们都会使用这个字段。在 Circle 类的内部, 该字段可以直接以 PI 来表示; 而在该类的外部, 若要指定该字段, 则类与字段名称都必须指定得很详细。其他不是 Circle 的方法如想要使用此字段, 则必须以 Circle.PI 才能访问到。

public 类字段基本上就是全局变量, 类字段的名称也是由定义它们的类的唯一名称来决定的。当不同的模块定义了相同名称的全局变量时, Java 不会遭遇到其他程序语言会发生的名词冲突问题。

## 类 method

跟类字段一样, 类 method 也使用 static 修饰符来声明:

```
public static double radiansToDegrees(double rads) { return rads * 180 / PI; }
```

上述的程序代码声明了一个名为 radiansToDegrees() 的 method, 它有一个 double 类型的参数与一个 double 类型的返回值。该 method 的主体非常简短, 只执行了一个简单的计算动作后就将结果返回。

和类字段一样, 类 method 是与类相关的, 而不是与对象相关。当从该类之外调用该类 method 时, 你必须同时指明类的名称与 method。例如:

```
// 半弧度值为 2.0 的角度值是多少呢?  
double d = Circle.radiansToDegrees(2.0);
```

如果你想要从某个类的内部调用该类的 method 时, 你不需指明该类的名称。然而, 不管在什么情况之下都指明类的名称, 这种做法是个很好的习惯, 它会让你更清楚地知道哪个类的 method 被调用了。

请注意, Circle.radiansToDegrees() method 使用了类字段 PI, 类 method 可以使用该类中的所有类字段与类 method, 但却不能使用任何的实例字段或实例 method, 因为类 method 与该类的实例是不相关的。换句话说, 虽然 radiansToDegrees() method 定义于 Circle 类, 但它不能使用任何的 Circle 对象。该类的实例字段与实例 method 是与 Circle 对象相关的, 而不是与类本身相关的。因为类 method 与该类的实例是没有关联的, 所以该类 method 不能使用任何的实例 method 或字段。

我们在之前已经讨论过, 类字段基本上就是全局变量。同样地, 类 method 也是全局 method 或全局函数。虽然 radiansToDegrees() 并不能作用于 Circle 对象, 但是因为当在处理与圆形有关的操作的时候, 它有时候是个有用的实用 method (utility method), 所以它被定义于 Circle 类中。在很多非面向对象的程序语言中, 所有的 method 或函数都是全局的。当然, 你可以只使用类 method 来编写复杂的 Java 程序, 然而, 这并不是

一个面向对象的程序设计方式，同时也没有利用Java语言所提供的强大功能。为了要写一个真正面向对象的程序，我们必须使用实例字段与实例method。

## 实例字段

任何没有使用 `static` 修饰符声明的字段都是实例字段：

```
public double r;    // 圆的半径
```

实例字段是与该类的实例相关联的，而不是与该类本身相关。因此，我们所创建的每一个 `Circle` 对象都拥有一个 `double` 类型的字段 `r`。在例子中，`r` 代表了圆的半径。因此，每一个 `Circle` 对象都会有一个与其他所有的 `Circle` 对象独立的半径。

在类定义里，对实例字段的引用只是通过它的名称。你可以察看实例 `method` `circumference()` 的主体，如此你就可以看到这样的一个例子。在该类外部的程序代码中，如果引用该实例 `method`，则必须在该实例 `method` 的名称前面加上该对象的名称。例如，如果变量 `c` 拥有一个对 `Circle` 对象的引用时，则可以使用 `c.r` 来引用圆的半径：

```
Circle c = new Circle(); // 创建一个Circle对象，将它存储于变量c中
c.r = 2.0;               // 赋一个值给它的实例字段r
Circle d = new Circle(); // 创建一个新的Circle对象
d.r = c.r * 2;           // 让此圆的半径为原先的两倍大
```

实例字段在面向对象程序设计中是相当重要的，实例字段拥有该对象的状态，而这些字段的值使得该对象不同于其他的对象。

## 实例 method

任何不是使用 `static` 关键字声明的方法都是实例 `method`。实例 `method` 只作用于类的实例（即对象），而不是该类本身。从实例 `method` 开始，面向对象程序设计开始变得有趣了。在例 3-1 中定义的 `Circle` 类，包含了两个实例 `method`，`area()` 与 `circumference()`，它们会计算与返回该 `Circle` 对象表示的圆的面积与周长。

如果你要从定义实例 `method` 类的外部来引用实例 `method`，就必须在该实例 `method` 的名称前面加上该实例的名称，例如：

```
Circle c = new Circle(); // 创建一个Circle对象并将它存储于变量c中
c.r = 2.0;               // 设定该对象的一个实例字段
double a = c.area();     // 调用一个该对象的实例method
```

如果你没有面向对象程序设计的经验，可能会对上述程序的最后一行感到陌生。我们并没有将它写成：



```
a = area(c);
```

而写成：

```
a = c.area();
```

这就是为什么我们称它为面向对象程序设计的原因。在这里的焦点 (focus) 是对象，而不是函数调用。这个小小的不同点，或许就是面向对象最重要的特点了。

重点是我们还没有将自变量传递给 `c.area()`，我们所操作的对象 `c` 在语法上存在一些隐含条件。再看一下例 3-1，你将在 `area()` method 的签名中发现相同的情况，它也没有自变量。现在再来看看 `area()` method 的主体，它使用了实例字段 `r`。因为 `area()` method 也在同一个类中，因此该 method 能调用 `r`。当任何 `Circle` 实例调用此方法时，便会引用该圆的半径。

另一个相当重要的一点是，`area()` 与 `circumference()` method 的主体都使用了类字段 `PI`。我们在先前就知道了类 method 只可以使用类字段与类 method，而不能使用实例字段或实例 method。实例 method 则不会受到这样的限制，它们可以使用类中的所有成员，不管它是否被声明为 `static`。

## 实例 method 的运作方式

请再看一下以下的程序代码：

```
a = c.area();
```

它会如何执行呢？没有参数的 method 该从何处获得数据来运算呢？事实上，`area()` method 是有一个参数的。所有的实例 method 在实现时都有一个隐含的参数，该参数并未出现在该 method 的签名中。此隐含的参数是 `this`，它包含了对调用 method 所在对象的引用，在我们的范例中，该对象就是 `Circle`。

这个隐含的 `this` 参数并不会出现在方法的签名中，因为通常情况下这种定义都是不需要的。每当 Java method 访问类中的实例字段时，就意味着它可以通过 `this` 参数来访问对象的字段。当实例 method 调用同一个类里的其他实例 method 时，同样的状况仍然成立。我在先前曾经说过，为了要调用实例 method，你必须在之前加上该对象的引用，而当该实例 method 被同一个类中的其他实例 method 所调用时，就不需要指明对象。在这样的情况下，该 method 会被认为是属于 `this` 对象。

你可以使用 `this` 关键字来明确地表示该 method 所调用的是它自己的字段和 / 或 method。例如：我们可以将 `area()` method 重新定义，并使用 `this` 来表示实例字段：

```
public double area() { return Circle.PI * this.r * this.r; }
```

上述的程序代码明确地使用类名来引用类字段PI。在这个简单的例子中,这样做其实是不需要的,但是在更复杂的情况下,你会发现在不需要this的地方放上this,会让你的程序更清楚易懂。

然而,在某些情况之下,this关键字是绝对必要的。例如,当方法中的method参数或局部变量的名称与该类中的某个字段名称相同时,你便必须使用this来指该字段,因为单独使用该字段名称时,所指的是method参数或局部变量。例如,我们可以将以下的method加入Circle类中:

```
public void setRadius(double r) {  
    this.r = r;        // 将自变量 (r) 赋给字段 (this.r)  
                        // 请注意我们不能写成 r = r  
}
```

最后要注意的是,实例method可以使用this关键字,但类method则不可以。这是因为类method与对象是不相关的。

### 实例method还是类method?

实例method是面向对象程序设计里一个相当重要的功能,但这并不表示类method就不重要了。在许多情况下,定义类method是比较适当的。例如,当使用Circle类时,你也许会发现到需要常常计算某个给定半径值的圆面积,但你又不想大费周章的去创建一个Circle对象来代表该圆。在这样的情况下,使用类method就会比较方便:

```
public static double area(double r) { return PI * r * r; }
```

在一个类里定义一个以上相同名称的method是绝对合法的,只要它们的参数不同即可。因为此版本的area() method是一个类method,所以它不会有隐含的this参数,这样就必须有一个指定圆半径的参数。这个参数让这个类method和与它同名的实例method有所不同。

再看另一个在实例method与类method之间作选择的例子,我们定义了一个名为bigger()的method,它会比较两个Circle对象并将拥有较大半径的对象返回。我们可以将bigger()写为如下所示的实例method:

```
// 比较隐含的“this”圆与作为自变量传递来的“that”圆  
// 并将较大者返回  
public Circle bigger(Circle that) {  
    if (this.r > that.r) return this;  
    else return that;  
}
```

我们也可以将bigger()实现为如下的类method:

```
// 比较圆 a 与圆 b 并将半径较大者返回
public static Circle bigger(Circle a, Circle b) {
    if (a.r > b.r) return a;
    else return b;
}
```

给定两个 Circle 对象，x 与 y，我们可以使用实例 method 或类 method 来判定哪一个圆比较大。然而，这两个方法的调用语法却是不同的：

```
Circle biggest = x.bigger(y);           // 实例 method, 也可以写作 y.bigger(x)
Circle biggest = Circle.bigger(x, y);    // 静态 method
```

这两个 method 都行得通，从面向对象设计的观点来看，这两个 method 都差不多。实例 method 是较为正式的面向对象，但它的调用语法受限于某种不对称性。在这样的情况之下，选择实例 method 还是类 method 仅只是一种设计决定。这取决于当时的情况，其中一种会是较自然的选择。

## 个案研究：System.out.println()

在这本书中，我们看到一个名为 System.out.println() 的 method，该 method 可以将输出显示在终端窗口或控制台上。我们还没解释过为什么此 method 的名称这么长，也没有说明该 method 中的两个句点所起的作用。现在你已经了解类、实例字段、类 method 与实例 method 了，那么要了解发生了什么事就比较容易了：System 是一个类，它有一个名为 out 的类字段，System.out 字段指向一个对象。对象 System.out 有个名为 println() 的实例 method。

## 对象的创建与初始化

我们已经介绍过字段与 method 了，接下来要介绍其他重要的类成员。构造函数 (constructor) 与初始化程序都是类成员，它们的工作是初始化类字段。

让我们来看看如何创建 Circle 对象：

```
Circle c = new Circle();
```

这个小括号在这里有什么用途呢？这让它看起来像在调用一个 method。事实上，这正是我们所要做的。Java 里的每一个类至少都会有一个与该类同名的构造函数，它的目的是对新对象执行必须的初始化操作。因为在例 3-1 里，我们并没有为 Circle 类定义构造函数，所以 Java 会给我们一个默认的构造函数，它没有任何的自变量，也不会执行特殊的初始化操作。



下面将介绍构造函数是如何执行的。使用new运算符为这个类创建一个新的但未初始化的实例。然后构造函数method会被调用，而新的对象会被隐含地传递给它(this引用，就像我们之前所看到的)，在括号内指明的自变量也会被显示地传递。构造函数可以使用这些自变量来做必要的初始化操作。

## 定义构造函数

我们可以为Circle对象做一些明确的初始化操作，所以让我们来定义一个构造函数。例3-2显示了新的Circle类定义，其中包含一个构造函数，能让我们指定新Circle对象的半径。构造函数也使用了this引用来区分拥有相同名称的method参数与实例字段。

例 3-2: Circle 类的构造函数

```
public class Circle {  
    public static final double PI = 3.14159; // 一个常量  
    public double r; // 一个定义圆半径的实例字段  
  
    // 构造函数：初始化半径字段  
    public Circle(double r) { this.r = r; }  
  
    // 实例method：依据半径来计算值  
    public double circumference() { return 2 * PI * r; }  
    public double area() { return PI * r*r; }  
}
```

当我们使用由编译器所提供的默认构造函数时，我们必须使用以下的方式来初始化半径：

```
Circle c = new Circle();  
c.r = 0.25;
```

如果我们使用此构造函数，初始化就会变成创建对象步骤的一部分：

```
Circle c = new Circle(0.25);
```

这里有一些关于命名、声明与编写构造函数时应注意的重要事项：

- 构造函数的名称一定要和类名相同。
- 和其他的method不一样的是，构造函数在声明时不可指定返回值的类型，即使是void也不可以。
- 构造函数的主体必须初始化this对象。
- 构造函数不可以返回this或其他任何的值。构造函数可以包含一个return语句，但不可以有返回值。

## 定义多个构造函数

在特殊的情况下你会想要使用最方便的方式,用不同的方法来设定对象的初始值。例如,我们会想要将圆的半径值设定为某个初始值或合理的默认值。因为我们的Circle类只有一个实例字段,当然不可能使用太多的方式来设定初始值。但在比较复杂的类里,定义多个构造函数是非常方便的,以下的例子是我们为Circle类定义了两个构造函数:

```
public Circle() { r = 1.0; }  
public Circle(double r) { this.r = r; }
```

为一个类定义多个构造函数是完全合法的,只要每一个构造函数有不同的参数列表就可以了。编译器会依你所给定的自变量个数与自变量类型来决定使用哪一个构造函数。这也是我们在第二章里所讨论到的方法重载(method overloading)的一个范例。

## 在一个构造函数中调用另一个构造函数

当类有多个构造函数时,this关键字会有一个特定的用法:在一个构造函数中调用同一个类中的另一个构造函数。换句话说,我们可以将之前的那两个Circle构造函数重写成以下的样子:

```
// 这是基本构造函数:将半径初始化  
public Circle(double r) { this.r = r; }  
// 此构造函数使用this()来调用上面的构造函数  
public Circle() { this(1.0); }
```

this()语法是一个method的调用,它会调用该类中的其他构造函数,当然,被调用的构造函数是由其自变量的个数与类型决定的。当许多的构造函数占用了大部分的初始化程序代码时,这就是相当有用的技术,因为它可避免程序代码的重复。如果只有一个参数的Circle()构造函数做了很多初始化的操作,则此例子便更能显示出这种做法的威力。

在使用this()时有个非常重要的限制:它只可以出现在构造函数的第一条语句里。当然,在它的后面可以接着此构造函数所需要的任何初始设定。这个限制的原因牵涉到对超类(superclass)构造函数method的自动调用,我们会在本章的稍后部分探讨。

## 字段默认值与初始化程序

并不是类的每一个字段都需要做初始化的操作。和全局变量不同的是,局部变量没有默认值,而且在未初始化之前是不能被使用的。但类字段是会自动地初始化成默认的初始值,false、'\u0000\'、0、0.0或null,依它们的类型而定。这些默认值都是由Java所指定的。如果该字段的默认值对你来说是不适当的,你可以自己指定一个不同的初始值。例如:

```
public static final double PI = 3.14159;  
public double r = 1.0;
```

字段声明与局部变量声明的语法相当类似,但对于如何处理它们的初始化表达式却完全不同。如第二章所说的,局部变量的声明是Java method 里的一个语句,当该语句被执行时,变量的初始化才会被执行。然而,字段的声明却不属于任何method,所以它无法被当作语句来执行,Java编译器会自动地产生实例字段的初始化程序代码,同时将它加入构造函数或类的构造函数中。初始化程序代码会依据在源代码中出现的顺序将它插入构造函数中,这表示字段初始化时可以使用该字段被声明之前使用的所有初始值。请看以下的一段程序代码,它包含了一个假设类中的一个构造函数与两个实例字段:

```
public class TestClass {  
    public int len = 10;  
    public int[] table = new int[len];  
  
    public TestClass() {  
        for(int i = 0; i < len; i++) table[i] = i;  
    }  
  
    // 类的其他部分省略……  
}
```

在这个情况下,会自动产生构造函数的程序代码如下:

```
public TestClass() {  
    len = 10;  
    table = new int[len];  
    for(int i = 0; i < len; i++) table[i] = i;  
}
```

如果构造函数一开始就使用了this()来调用其他的构造函数,则字段初始化程序代码就不会出现在第一个构造函数里,而是会出现在this()所调用的构造函数里。

如果实例字段是在构造函数method中被初始化的话,那么,类字段是在哪里被初始化的呢?因为即使没有任何的类实例被创建,这些字段还是和类有关联的,所以在构造函数被调用之前,它们必须被初始化。为了实现这一点,Java编译器会自动为每一个类产生一个类初始化方法(class initialization method)。类字段是在此method的主体里被初始化,因此在类第一次被使用时(当该类第一次被JVM加载时)(注2),类字段的初始化一定会被执行一次。跟实例字段初始化一样,类字段初始化表达式会依照它在源代码内出现的顺序将其插入类初始化method中。这就表示类字段的初始化表达式可以利

---

注2: 为类C编写类的初始化程序且此程序调用了创建C的实例的另一个类的method,这在实际上是有可能的。在这个创造出来的递归范例里,C的实例在类C被完全初始化之前就已经创建好了。然而,这样的情况在实际中并不常见。



用声明在它之前的类字段。类初始化方法是一个被隐藏起来的内部方法 (internal method)，它对程序员来说是不可见的。在类文件里，它的名字叫 `<clinit>`。

## 初始化模块

到目前为止，我们已经知道对象可以通过它们的字段初始化表达式以及在它们构造函数 `method` 中的程序代码来加以初始化。类有一个类初始化 `method`，它和构造函数一样，但我们不能像定义构造函数那样明确地定义此 `method` 的主体。但是，Java 的确允许我们以所谓的静态初始化程序 (static initializer) 来针对类字段的初始化编写任意的程序代码。静态初始化程序只是在 `static` 关键字后加上一个由大括号所括起的代码段。静态初始化程序可以出现在类定义中任何允许出现在字段或 `method` 定义的地方。例如，请看以下的程序代码，它会对两个类字段做初始化操作：

```
// 我们可以使用三角函数画出圆的轮廓
// 但是，这样做的速度太慢了，所以我们预先计算出一组值
public class TrigCircle {
    // 下面是静态查询表以及这些表本身的简单初始化程序
    private static final int NUMPTS = 500;
    private static double sines[] = new double[NUMPTS];
    private static double cosines[] = new double[NUMPTS];

    // 这里是写入数组的静态初始化程序
    static {
        double x = 0.0;
        double delta_x = (Circle.PI/2)/(NUMPTS-1);
        for(int i = 0, x = 0.0; i < NUMPTS; i++, x += delta_x) {
            sines[i] = Math.sin(x);
            cosines[i] = Math.cos(x);
        }
    }
    // 省略该类的其他程序代码……
}
```

一个类可以有多个静态初始化程序。每一个初始化程序模块的主体，要与所有的静态字段初始化表达式一起放入类的初始化 `method` 中。静态初始化程序和类 `method` 一样，不可以使用 `this` 关键字或该类中任何的实例字段或实例 `method`。

在 Java 1.1 以及之后的版本里，类也可以有实例初始化程序。实例初始化程序与静态初始化程序一样，除了它是对类做初始化而不是对对象做初始化之外。一个类可以有多个实例初始化程序，它们可以出现在允许出现字段或方法定义的地方。每一个实例初始化程序的主体会与所有的字段初始化表达式一起被放入该类的每一个构造函数的最开始处。实例初始化程序看起来就像静态初始化程序一样，除了它不能使用 `static` 关键字之外。换句话说，实例初始化程序是一个由大括号括起的任意 Java 程序块。

实例初始化程序可以初始化数组或其他需要复杂初始化操作的字段。有时候它们是非常有用的，因为它们将初始化程序代码紧接着该字段，而不是将它分散在构造函数method中。例如：

```
private static final int NUMPTS = 100;
private int[] data = new int[NUMPTS];
{ for(int i = 0; i < NUMPTS; i++) data[i] = i; }
```

然而，实际上，像这样的实例初始化程序的使用已经很少见了。在Java 1.1中，实例初始化程序可以用来支持匿名内部类（anonymous inner class），但不可以用来定义构造函数（匿名内部类会在稍后的“嵌套类型”一节中介绍）。

## 对象的撤消与终止

我们已经知道在Java里如何创建新对象与初始化值，现在要来讨论对象生命周期的另一个终端，了解对象是如何被撤消与终止。终止（finalization）是与初始化完全相反的。

在Java里，当对象不再被需要的时候，该对象所占用的内存会被自动地回收。这是通过内存回收（garbage collection）程序来完成的。内存回收这样的技术已经在像是Lisp这样的程序语言里使用许多年了。C与C++的程序员必须调用free()函数或delete运算符来做内存回收操作，而在Java里，你不需要考虑撤消对象，这也使得Java格外令人喜欢。这样的特点也使得Java程序中的程序错误比起那些不支持自动内存回收的程序语言来说减少了许多。

## 内存回收（garbage collection）

Java解释器很清楚地知道它取得了哪些对象与数组，它也知道各个局部变量调用了哪些对象和数组以及那些对象或数组又调用了哪些其他的对象或数组。因此，解释器当然会知道对象何时已不再被其他的变量或对象所使用。当解释器找到这样的对象时，它知道可以安全地收回该对象占用的内存。内存回收也可以侦测并撤消对象之间的循环（cycle）。循环是指某些对象之间有相互的引用，但并没有被其他操作中的对象所引用。任何这样的循环都会被回收。

不同的VM在处理内存回收的方式上本来就会有所不同。这是很合理的，内存回收是以一个低优先权的线程来执行的，所以在没有其他的事情要做时才会执行它大部分的操作，例如在等待用户输入数据的空闲时间。内存回收被当成高优先权来执行的唯一情况（也就是实际上真的会使系统崩溃的时候），就是可用的内存变得非常非常少的时候。但这并不会常常发生，因为这个低优先权的线程会在后台不停地清除不需要的对象。

## Java 中的内存泄漏

Java 所支持的内存回收可大幅降低内存泄漏 (memory leak) 的发生率。内存泄漏会发生在内存已被分配但却没有被回收时。刚开始时,你也许会认为内存回收可以预防内存泄漏的原因,是因为它会回收所有不再被使用到的对象。然而,在 Java 里,如果一个对不再被使用的对象的有效(但不再被使用)引用被闲置时,内存泄漏的情形仍然会发生。例如,当一个 method 执行了很久(甚至会一直执行下去),在那个 method 中的局部变量所保留的对象引用可能会比实际所需的时间还长。以下的程序代码可作说明:

```
public static void main(String args[]) {
    int big_array[] = new int[100000];

    // 使用 big_array 做一些运算并取得结果
    int result = compute(big_array);

    // 我们不再需要 big_array 了。当没有其他的引用指向它时,它便会被回收。
    // 因为 big_array 是一个局部变量,它会引用该数组,直到此 method 结束为止。
    // 但这个 method 不会结束,因此,我们自己必须显式撤消此引用,
    // 以让内存回收程序知道可以回收该数组
    big_array = null;

    // 此为无穷循环,用来处理用户的输入
    for(;;) handle_input(result);
}
```

内存泄漏也会发生在你使用哈希表 (hash table) 或其他类似的数据结构来将某个对象关联到另外一个对象时。即使这两个对象已不再被需要了,但它们之间的关联性仍会被保留在哈希表里,直到该哈希表被回收时,该对象才会跟着被回收。如果哈希表的生命周期比该对象长的话,内存泄漏的情况就会发生。

要避免内存泄漏最重要的就是,当拥有这些引用的对象一直都是存在的情形下且该对象不再被需要时,就必须将该对象的引用设定为 null 值。有个常见的内存泄漏来源,就是在 Object 数组被用来代表由对象组成的 collection 数据结构里。一般会使用独立的 size 字段来记录目前数组的哪些元素是合法的。当从集合里移走一个对象时,只递减 size 字段是不够的,你必须要将适当的数组元素值设定为 null,以便过时的对象引用不再存在。

## 对象的终止

Java 中的终止函数 (finalizer) 与构造函数是完全相反的,就像构造函数会执行对象的初始化操作一样,终止函数则会执行对象的清除或结束操作。打开内存回收会自动地释放对象所占用的内存空间,但对象也会占有其他不同类型的资源,如打开文件与网络连接等。内存回收对于这种资源就无法做回收操作,所以你必须写一个终止方法来处理关



闭已打开的文件、结束网络连接、删除临时文件等操作。使用 native method 对类来说是可以的,而这些类可能会需要一个终止函数将Java内存回收的无法处理的资源(包括内存)给释放掉。

终止函数是一个实例method,它没有任何的自变量或返回值。每一个类只能有一个终止函数,而且它必须被命名为`finalize()`(注3)。终止函数可以抛出任何类型的异常(exception)或错误(error),但当终止函数是自动地被内存回收程序所调用时,它所抛出的异常与错误都将会被忽略,而只把这些异常或错误看作是让终止函数方法返回的标志。终止函数一般都被声明为`protected`类型(我们还没讨论到),但有时候也会声明为`public`类型。以下是终止函数的范例:

```
protected void finalize() throws Throwable {  
    // 调用超类的终止函数  
    // 我们尚未讨论到超类与它的语法  
    super.finalize();  
  
    // 删除一个我们之前所使用的临时文件  
    // 如果文件不存在或tempfile为null,则会抛出一个异常,但该异常会被忽略  
    tempfile.delete();  
}
```

这有一些关于终止函数的重要事项:

- 如果对象有一个终止函数,则对象不再被使用(或无法取得)时,在内存回收程序要收回它之前,必须调用终止函数。
- Java不保证内存回收的操作一定会发生,对象回收也没有一定的顺序性。因此,Java也不确定终止函数何时会被调用,而且不能保证终止函数被调用的顺序以及哪个线程将执行终止函数。
- Java解释器可以在不回收所有存在对象的情况下结束执行,因此有些终止函数将不会被调用。在这样的情况下,如网络连接等资源会由操作系统来关闭及回收。但请注意,如果终止函数要删除一个文件但却没有去执行这个动作,则操作系统将不会去删除该文件。

为了确保在虚拟机结束之前一定会执行特定动作,Java 1.1中提供了Runtime方法`runFinalizersOnExit()`。但不幸的是,此method会导致死锁(deadlock)且相当的不安全。在Java 1.2里不建议使用,但在Java 1.3以及之后的版本里,

---

注3: C++程序员应该注意到,虽然Java的构造函数方法的命名方式和C++的构造函数一样,但Java的终止函数方法的命名方式却和C++的撤消方法不一样。就如我们将看到的一样,它们的运行方式和C++的撤消方法并不十分的相似。

Runtime方法`addShutdownHook()`可以在Java解释器结束前安全地执行任何的程序代码。

- 在调用终止函数之后,对象并不是立刻就会被释放出来。这是因为终止函数会在某个地方存储 `this` 指针而让对象复活 (resurrect), 所以该对象又可以再次获得引用。因此,在调用 `finalize()` 之后,实际回收对象之前,内存回收程序必须再一次确认该对象已未被引用。但即使对象复活,终止函数也不会再被调用一次,且对象复活后绝对不会去做任何有用的事情——它只是对象终止过程中的一个很奇怪的现象而已。
- `finalize()` method 是个实例 method, 且终止函数是在实例上运作。对于类的终止并没有提供相似的机制。

实际上,应用级的类不太需要 `finalize()` 方法。然而,在写 Java 类且使用了 native method 来作为平台间的接口时,终止函数就变得非常的有用。在这样的情况下,使用 native `finalize()` 方法来执行回收的动作可以回收不在Java内存回收控制之下的其他资源。

此外,因为终止函数在何时执行以及是否执行的不确定性,所以最好避免终止函数之间的依赖性。例如,包含对网络 socket 引用的类应该要定义一个公共的 `close()` method, 它会调用 socket 的 `close()` method。这样,当类的用户在使用它时,就可以调用 `close()` 来确保网络连接被关闭。然而,你也可以定义 `finalize()` method 来作备用,以便在使用这个类的用户忘了调用 `close()` 时让未被关闭的实例能被回收。

## 子类与继承

之前我们所定义的 `Circle` 是一个很简单的类,该类与各个对象之间只有半径这个特性不同而已。现在假设我们要表示出某个圆的尺寸与位置,例如,一个半径为 1.0、圆心在直角平面坐标系上为 (0,0) 的圆,与半径为 1.0、圆心在 (1,2) 的圆是两个不同的圆。为了区分这两个圆,我们需要一个新的类,我们将其称为 `PlaneCircle`, 同时我们保留了 `Circle` 类中的功能,并另外加上一个能够表示圆的位置的功能。于是我们把 `PlaneCircle` 定义为 `Circle` 的一个子类,如此 `PlaneCircle` 就会继承它的父类 `Circle` 的所有字段与 method。加入新功能以产生新的子类的能力,或者说是扩展父类以产生子类的能力,是面向对象程序设计最核心的概念。

## 类的扩展

例 3-3 即是我们将 `PlaneCircle` 变为 `Circle` 类的一个子类的例子:

## 例 3-3: 扩展 Circle 类

```

public class PlaneCircle extends Circle {
    // 我们会自动继承 Circle 的字段与 method,
    // 所以只要在这里加些新的功能就可以了
    // 新的实例字段, 用来存储圆心坐标
    public double cx, cy;

    // 用来初始化新字段的新构造函数方法
    // 它使用了特殊的语法来调用 Circle() 构造函数
    public PlaneCircle(double r, double x, double y) {
        super(r);          // 调用父类的构造函数, Circle()
        this.cx = x;        // 初始化实例字段 cx
        this.cy = y;        // 初始化实例字段 cy
    }

    // area() 与 circumference() 方法是继承自 Circle
    // 有个新的实例 method 会检查某个点是否在圆内
    // 请注意, 它使用了继承的实例字段 r
    public boolean isInside(double x, double y) {
        double dx = x - cx, dy = y - cy;          // 到圆心的距离
        double distance = Math.sqrt(dx*dx + dy*dy); // 勾股定理
        return (distance < r);                     // 返回 true 或 false
    }
}

```

请注意, 例 3-3 的第一行使用了 extends 关键字, 这个关键字在告诉 Java, PlaneCircle 是 Circle 类的扩展, 或者说 PlaneCircle 是 Circle 的子类, 这表示它继承了 Circle 类的字段与 method (注 4)。isInside() method 的定义用到了字段继承的特性, 此 method 使用了字段 r (由 Circle 类所定义) 就如同它在 PlaneCircle 里面所定义的一样。PlaneCircle 也继承了 Circle 的 method, 因此, 如果我们有个变量 pc 引用到 PlaneCircle 对象, 我们可以写成:

```
double ratio = pc.circumference() / pc.area();
```

这样做就好像 area() 与 circumference() method 是在 PlaneCircle 里所定义的一样。

子类的另一个特性就是每一个 PlaneCircle 对象都是完全合法的 Circle 对象。如果 pc 引用 PlaneCircle 对象, 我们就可以将它赋值给一个 Circle 变量, 而不管它的额外的功能。

```

PlaneCircle pc = new PlaneCircle(1.0, 0.0, 0.0); // 圆心在坐标原点的单位圆
Circle c = pc; // 不需使用强制转换的方式就可将 pc 赋值给一个 Circle 变量

```

注 4: C++ 程序员应该注意到, Java 中的 extends 和 C++ 中的: 是相同的, 在 C++ 中, 这两者都是用来指出该类的超类。



将 `PlaneCircle` 对象赋值给 `Circle` 变量并不需要做强制转换的操作, 就如同我们在第二章“引用类型的转换”一节所讨论过的, 像这样的放大转换永远都是合法的。而 `Circle` 类型的变量 `c` 所拥有的值仍然是合法的 `PlaneCircle` 对象, 但编译器本身并不肯定这一点, 因此它并不允许我们在不使用强制转换的情况下去执行反方向(缩小转换)的非强制转换:

```
// 缩小转换需要强制转换 (同时由 JVM 在执行运行时检查)
PlaneCircle pc2 = (PlaneCircle) c;
boolean origininside = ((PlaneCircle) c).isInside(0.0, 0.0);
```

## final 类

当一个类使用 `final` 修饰符来声明时, 就表示它是无法被扩展的, 或者说我们不能够用它来定义子类。`java.lang.String` 就是一个 `final` 类的例子。将一个类声明为 `final` 可以避免对该类进行的不必要的扩展。如果你调用了一个 `String` 对象的 `method`, 而且你也知道该 `method` 只有在 `String` 类里才有定义的, 即使 `String` 是通过一些外在来源被传递给你。因为 `String` 是被定义为 `final`, 所以没有任何的类可以是它的子类, 同时也无法改变其 `method` 的意义或行为。

将类声明为 `final`, 允许编译器在调用类的 `method` 时进行一些优化。我们将在本章稍后讨论关于 `method` 的覆盖 (overriding)。

## 超类、对象与类层次

在我们的例子里, `PlaneCircle` 是 `Circle` 类的一个子类。我们也可以说 `Circle` 类是 `PlaneCircle` 类的超类。我们可以使用 `extends` 子句来指定一个类的超类:

```
public class PlaneCircle extends Circle { ... }
```

你所定义的每一个类都会有一个超类。如果你没有以 `extends` 子句来指定超类, 则该类的超类就是 `java.lang.Object` 类。基于以下两个理由, 我们将 `Object` 视为一个特殊的类:

- 它是 Java 里唯一没有超类的类。
- 所有的 Java 类都继承了 `Object` 的 `method`。

因为每一个类都会有一个超类, 所以在 Java 中的类会构成一个类层次 (class hierarchy) 的结构, 并且可以被表示成以 `Object` 为根的树形结构。图 3-1 所显示的就是一个包含 `Circle` 与 `PlaneCircle` 类以及一些 Java API 标准类的层次结构图。

## 子类的构造函数

请再看一次例 3-3 的 `PlaneCircle()` 构造函数方法：

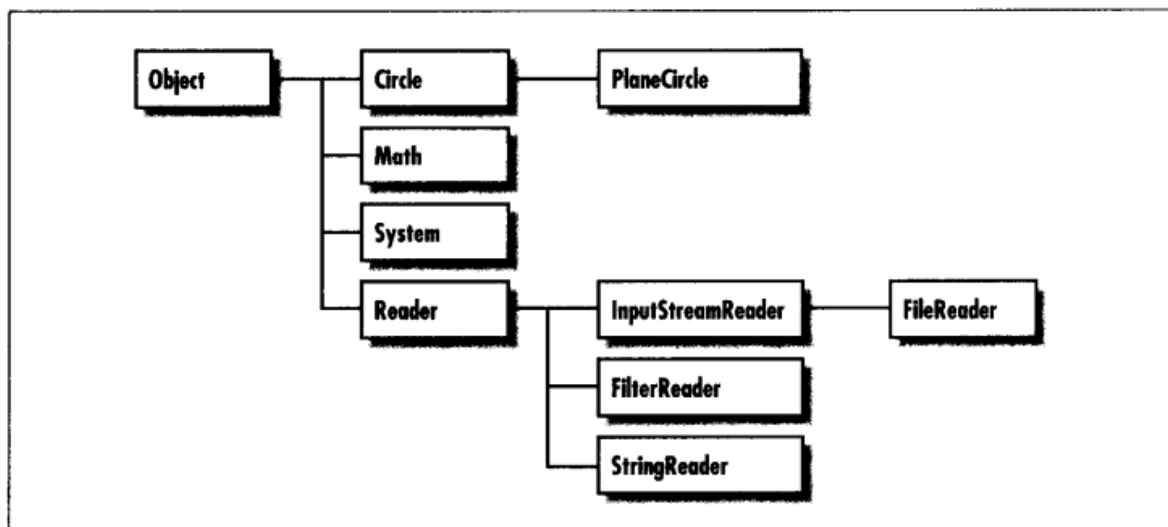


图 3-1：类层次示意图

```
public PlaneCircle(double r, double x, double y) {  
    super(r);        // 调用超类的构造函数Circle()  
    this.cx = x;      // 初始化实例字段 cx  
    this.cy = y;      // 初始化实例字段 cy  
}
```

此构造函数会对 `PlaneCircle` 类新定义的 `cx` 与 `cy` 字段做明确地初始化设定，但它必须依赖超类 `Circle()` 的构造函数来初始化该类所继承的字段。为了要调用超类的构造函数，我们的构造函数会调用 `super()`。`super` 是 Java 的保留字。它的一种用途就是在子类的构造函数方法中调用了超类构造函数方法。这样的使用方法和使用 `this()` 类似，`this()` 能从某个类的构造函数中调用同一个类的其他构造函数。使用 `super()` 来调用构造函数和使用 `this()` 来调用构造函数有相同的限制：

- `super()` 这种用法只能用在构造函数中。
- 对超类构造函数的调用必须出现在构造函数的第一个语句里，甚至必须出现在局部变量声明之前。

传递给 `super()` 的自变量必须与超类构造函数的自变量相匹配。如果超类定义了一个以上的构造函数时，`super()` 可以调用所有构造函数中的任何一个，但会依所传递的自变量来选择最适当的一个。

## 构造函数链与默认构造函数

Java 会保证当类的实例被创建时, 该类的构造函数一定会被调用到; 它也会保证当该类的子类的实例被创建时, 该构造函数也一定会被调用。为了要保证第二点, Java 必须确保每个构造函数都可以调用它的超类的构造函数方法。因此, 如果构造函数内的第一行语句并没有使用 `this()` 或 `super()` 来调用其他的构造函数时, 则 Java 会自行插入 `super()` 调用, 也就是说它会调用无自变量的超类构造函数。如果超类没有不带自变量的构造函数时, 那么这个默认的调用就会造成编译错误。

当我们创建一个 `PlaneCircle` 类的新实例时, 想想看, 会发生什么事。首先, `PlaneCircle` 的构造函数会被调用, 此构造函数会调用 `super(r)` 来调用 `Circle` 的构造函数, 同时构造函数 `Circle()` 会调用 `super()` 来调用它的超类的构造函数 `Object`。会先从 `Object` 构造函数的主体开始执行, 当它执行完毕且返回时, 便会接着执行 `Circle()` 构造函数的主体。最后, 当对 `super(r)` 的调用也返回时, `PlaneCircle()` 构造函数中的其他语句就会接着被执行。

以上的说明只是要告诉你构造函数的调用是相互链接的 (chained)。每当创建一个对象时, 就会调用一连串的构造函数, 从子类到超类, 一直到位于类层次结构根部的 `Object` 类为止。因为超类的构造函数都会在子类构造函数的第一行语句里被调用, 所以 `Object` 构造函数的主体总是会第一个被执行, 接下来就是执行它的子类的构造函数, 顺着类层次结构一直往下走, 最后才轮到被实例化的类。当构造函数被调用时, 它的超类的字段也会同时被初始化。

### 默认构造函数

上面对构造函数的链接的说明中少了一个很重要的部分, 如果构造函数没有调用它的超类的构造函数时, Java 就会默认调用其超类的构造函数。但是如果该类根本就没有声明构造函数呢? 在这样的情况下, Java 会默认地帮该类定义一个构造函数。这个默认的构造函数除了会调用其超类的构造函数之外, 其他的事都不会去做。例如, 如果我们没有替 `PlaneCircle` 类声明一个构造函数, 那么 Java 会默认地插入以下这个构造函数:

```
public PlaneCircle() { super(); }
```

如果超类 `Circle` 没有声明无自变量的构造函数, 则在此构造函数 `PlaneCircle()` 中自动插入的对默认构造函数 `super()` 的调用会造成编译错误。一般来说, 如果类没有定义无自变量的构造函数, 那么它的所有子类所定义的构造函数都必须加入所需的自变量以明确地调用超类的构造函数。



如果一个类没有声明任何的构造函数，则它会被给定一个默认且无自变量的构造函数。被声明为`public`的类将会把构造函数也定义为`public`，而提供给所有其他类的默认构造函数的声明则不包含任何显式修饰符（visibility modifier），像这样的构造函数拥有默认显式修饰符（显式的概念会在本章稍后加以解释）。如果你创建了一个`public`类，但又不希望它能公开地被实例化，那么你应该至少声明一个非`public`类型的构造函数，以避免默认的`public`构造函数被插入。若某个类永远都不想被实例化（如`java.lang.Math`或`java.lang.System`），就必须定义一个`private`类型的构造函数，像这样的构造函数将永远无法从该类的外部加以调用，但它同时防止了默认构造函数的自动插入。

### 终止函数链？

你可能会做这样的假设，既然Java链接了构造函数方法，那它也应该自动地链接对象的终止函数方法。换句话说，你也可能会假设类的终止函数会自动地调用其超类的终止函数。事实上，Java并不会这样做。当你在写一个`finalize()`方法时，你必须显式地调用其超类的终止函数（即使你知道该超类没有终止函数时，你也必须这么做，因为超类的实现在之后也许会被加入终止函数）。

就如我们在本章稍早的范例里所看到的终止函数，你可以使用`super`关键字的语法来调用超类的方法：

```
// 调用超类的终止函数方法
super.finalize();
```

当我们讨论到`method`的覆盖时，会更详细地讨论这个语法。事实上，终止函数是很少用到的，所以终止函数链也会很少见到。

### 隐含超类字段

考虑这个例子，假设我们的`PlaneCircle`类需要知道圆心到坐标原点(0,0)的距离，我们可以加入另外的实例字段来记录这个值：

```
public double r;
```

为构造函数加入以下的程序代码来计算字段的值：

```
this.r = Math.sqrt(cx*cx + cy*cy); // 勾股定理
```

但请稍等，这个新字段`r`与`Circle`超类的半径字段`r`的名称相同。当这样的情形发生时，我们说`PlaneCircle`类的字段`r`隐含了`Circle`的字段`r`（当然，这是个人为的例

子:新字段应该被称为distanceFromOrigin。虽然你应该试着要去避免这种情况发生,但子类字段有时候确实会将超类的字段给隐藏起来)。

使用此PlaneCircle的新定义时,表达式r与this.r都是指PlaneCircle中的字段。那我们要如何才能涉及到Circle的r字段呢?可以使用super关键字的特殊语法:

```
r          // 指的是 PlaneCircle 中的字段
this.r     // 指的是 PlaneCircle 中的字段
super.r    // 指的是 Circle 中的字段
```

另一个用来引用隐含字段的方法是将this(或该类的任何实例)强制转换成适当的超类,然后再访问其字段:

```
((Circle) this).r    // 引用 Circle 类的字段 r
```

当你需要引用一个隐含字段,而此字段并不是直接在超类中被定义时,强制转换的技术就特别有用。例如,假设A、B与C类都定义了一个相同名称的字段x,且C是B的子类,B是A的子类,然后,在类C的方法里,你可以用以下的方式来引用这些不同的字段:

```
x                // 类 C 中的字段 x
this.x           // 类 C 中的字段 x
super.x          // 类 B 中的字段 x
((B)this).x      // 类 B 中的字段 x
((A)this).x      // 类 A 中的字段 x
super.super.x    // 不合法,无法引用类 A 中的字段 x
```

你无法使用super.super.x来引用某个超类的超类中的隐含字段x。这是不合法的语法。

同样地,如果类C有一个实例c,则你可以用以下的方式来引用这三个名为x的字段:

```
c.x              // 类 C 中的字段 x
((B)c).x         // 类 B 中的字段 x
((A)c).x         // 类 A 中的字段 x
```

到目前为止,我们已经讨论过实例字段了。类字段也是可以隐含的,你可以使用相同的super语法来引用字段的隐含值,但你并不需要这么做,因为你一定可以使用该字段的名称来引用类字段的值。假设PlaneCircle的实现者决定要更改Circle.PI的值,因为他觉得原先的设定值不够精确,他可以定义他自己的类字段PI:

```
public static final double PI = 3.14159265358979323846;
```

现在,在PlaneCircle中的程序代码可以通过PlaneCircle.PI或表达式PI来使用这个更为精确的数字了,他也可以通过表达式super.PI与Circle.PI来引用旧的数

值。请注意，PlaneCircle所继承的area()与circumference()方法是定义于Circle类，所以它们要使用Circle.PI，即使该值已经被PlaneCircle.PI隐含了。

## 覆盖（overriding）超类 method

当一个类定义的实例method与它的超类中的method使用了相同的名称、返回类型与参数时，则该类的方法会覆盖（override）其超类的method。当该method被该类的对象所调用时，被调用的是新定义的method，而不是超类的method。在Java 5.0以及之后的版本中，覆盖method的返回值类型可以用覆盖method的返回值类型的子类来取代相同的类型。这就是我们在第二章所介绍的“协变返回类型”。

method覆盖（overriding）在面向对象程序设计里是一项很重要且有用的技术。PlaneCircle并没有覆盖Circle所定义的方法，但现在假设我们定义了另一个名为Ellipse的Circle的子类（注5）。在这种情况下，要覆盖继承自Circle的area()与circumference()方法就很重要了，因为这些用来计算圆面积与周长的方程式并不适用于椭圆。

接下来要介绍的method覆盖只考虑实例method。因为类method和实例method有很大的不同，能介绍的部分不多。就和字段一样，类method可以被子类所隐含，但是不可以被覆盖。就像在本章的前面所提到的，在调用类method时，在该method名称前面加上定义该method的类名，是一个相当好的程序编写方式。如果依据此方法来写程序，则这两个method的名称便不会相同，所以并没有任何东西会被隐含。然而，使用类method来隐含实例method是不合法的。

在我们要更进一步地讨论method覆盖之前，你应该了解method覆盖与method重载（method overloading）之间的差异。就如我们在第二章里所讨论过的，方法重载是指将多个method（在同一个类中）定义为名称相同，但却拥有不同的参数列表。这是和method覆盖差异极大的地方，所以千万不要搞混了。

## 覆盖并不是隐含

虽然在很多方面Java都认为类字段与类method是相似的，但method覆盖却和字段隐含却是完全不同的。你可以通过强制转换来将对象转成相应超类的实例来引用隐含的字段，

---

注5：数学家可能会对此有所争辩，因为所有的圆都是椭圆，那么Ellipse应该是超类而Circle是子类。但实际的工程师可能会用较少的实例字段来代表圆，所以Circle对象不应该从Ellipse继承一些不必要的字段。无论如何，这是一个有用的例子。



但却不能使用这样的技术来调用被覆盖的实例method。以下的程序代码说明了这个重大的差异：

```
class A {                                // 定义一个名为A的类
    int i = 1;                            // 一个实例字段
    int f() { return i; }                 // 一个实例method
    static char g() { return 'A'; }       // 一个类method
}

class B extends A {                      // 定义一个A的子类
    int i = 2;                            // 隐含类A中的字段 i
    int f() { return -i; }                // 覆盖类A中的实例method f
    static char g() { return 'B'; }       // 隐含类A中的类method g()
}

public class OverrideTest {
    public static void main(String args[]) {
        B b = new B();                    // 创建一个B类型的新对象
        System.out.println(b.i);           // 引用 B.i, 打印结果为 2
        System.out.println(b.f());         // 引用 B.f(), 打印结果为 -2
        System.out.println(b.g());         // 引用 B.g(), 打印结果为 B
        System.out.println(B.g());         // 这是调用 B.g() 的较好的方式

        A a = (A) b;                      // 将 b 强制转换成类 A 的一个实例
        System.out.println(a.i);           // 引用 A.i, 打印结果为 1
        System.out.println(a.f());         // 引用 B.f(), 打印结果为 -2
        System.out.println(a.g());         // 引用 A.g(), 打印结果为 A
        System.out.println(A.g());         // 这是调用 A.g() 的较好的方式
    }
}
```

一开始你可能会对 method 覆盖与字段隐含之间的差异感到相当的惊讶，但只要稍微想一想，就可以弄清楚它的目的。假设我们正在处理一些 Circle 与 Ellipse 类型的对象，由于要记录这些圆与椭圆的状态，我们把它们放在一个 Circle[] 类型的数组里（我们能这么做是因为 Ellipse 是 Circle 的子类，所以所有的 Ellipse 对象都是合法的 Circle 对象）。当我们逐一处理数组里的元素时，我们不必知道或在意该元素实际上是 Circle 还是 Ellipse。但我们必须关心的是，当我们在调用数组元素的 area() method 时，它所计算出来的是不是一个正确的值。换句话说，当该对象是一个椭圆形的时候，我们不会使用计算圆面积的公式来计算椭圆形的面积。基于这种考虑，我们应该不会对 Java 的 method 覆盖和字段隐含之间的不同处理方式感到惊讶了。

## 动态 method 查找

如果我们有一个拥有 Circle 与 Ellipse 对象的 Circle[] 数组，编译器如何能知道数组里的对象调用的是 Circle 类的 area() 还是 Ellipse 类的 area() 呢？事实上，编译器并不知道，也无法知道。然而，编译器了解它自己无法知道，所以在执行时刻会使

用动态 method 查找 (dynamic method lookup) 来产生程序代码。当解释器执行该程序代码时, 就会为数组里的每一个对象寻找最适合的 `area()` method。也就是当解释器在处理 `o.area()` 表达式时, 它会自动地找出变量 `o` 引用的特定对象的 `area()` method, 它不会只使用和变量 `o` 的类型静态相关的 `area()` method。这种动态 method 查找的过程有时也被称为虚拟方法调用 (virtual method invocation) (注 6)。

### final method 与静态 method 查找

虚拟方法调用是相当快的, 但在运行时不需要动态查找的情况之下, method 调用会更快。很幸运, Java 并不是一直需要使用动态 method 查找, 尤其是如果有个 method 被声明为 `final`, 就表示此 method 的定义是最终的, 它不可以被任何的子类覆盖。如果某个 method 无法被覆盖, 编译器就会知道此 method 的定义是唯一, 就不会需要动态 method 查找 (注 7)。此外, 所有 `final` 类中的 method 也不可被覆盖。就如本章稍早所讨论过的, `private` method 不能被子类继承, 同时也无法被覆盖 (亦即所有的 `private` method 都是 `final` 类型的)。最后, 类 method 的行为跟字段一样 (亦即它们可以被子类继承但无法被覆盖)。总地来说, 类中所有的 method 若被声明为 `final`, 就和所有声明为 `final`、`private` 或 `static` 的 method 一样, 在被调用时都不需要使用到动态 method 查找。这些 method 在运行时也常被实时 (JIT) 编译器或其他类似的优化工具当作是程序内嵌 (inlining) 的候选。

### 调用被覆盖的 method

我们已经看到了 method 覆盖与字段隐含之间的主要差异了。不过, Java 用来调用被覆盖 method 的语法, 与访问隐含字段的语法是相当类似的, 两者都使用了 `super` 关键字。以下的程序代码可以解释这一点:

```
class A {
    int i = 1;                // 一个被子类 B 所隐含的实例字段
    int f() { return i; }     // 一个被子类 B 所覆盖的实例 method
}

class B extends A {
    int i;                    // 此字段隐含了 A 中的 i 字段
    int f() {                 // 此 method 覆盖了 A 中的方法 f()

```

注 6: C++ 程序员应该注意到, 动态 method 查找就是 C++ 里执行的 virtual 函数。Java 与 C++ 之间的重要差异就是 Java 并没有 `virtual` 这个关键字。在 Java 里, method 都默认为 `virtual`。

注 7: 依此来看, `final` 修饰符和 C++ 里的 `virtual` 修饰符是对立的。Java 中所有非 `final` 的 method 都是 `virtual` 类型的。

```
        i = super.i + 1;           // 可用此方法来访问 A.i
        return super.f() + i;     // 可用此方式来调用 A.f()
    }
}
```

请回想之前你在使用 `super` 来引用一个被隐含的字段时，就如同将 `this` 强制转换为其超类的类型，然后再通过它来访问这个字段。然而，使用 `super` 来调用一个被覆盖的方法与强制转换 `this` 是不一样的。换句话说，上述的程序代码中的表达式 `super.f()` 是与 `((A)this).f()` 不同的。

当解释器使用 `super` 语法来调用一个实例 `method` 时，就会执行修正过的动态 `method` 查找。第一步跟正常的动态 `method` 查找一样，会判断该类所调用的方法到底是在哪个类中，在正常的情况下，对某个 `method` 定义的动态查找会从当前类开始。当 `method` 通过 `super` 语法被调用时，查找就会从当前类的超类开始。如果超类直接实现了该 `method`，则该 `method` 便会被调用。如果超类继承了该 `method`，则该 `method` 的继承便会被调用。

请注意，`super` 关键字调用的最直接地被覆盖的 `method`。假设类 `A` 有一个子类 `B`，而 `B` 有一个子类 `C`，这三个类中都定义了相同名称的 `f()` `method`。`method C.f()` 可以调用 `method B.f()`，而 `B.f()` 已经被 `C.f()` 覆盖了，但可以通过 `super.f()` 来调用 `B.f()`。但 `C.f()` 却无法直接调用 `A.f()`，因为 `super.super.f()` 并不是合法的 `Java` 语法。当然，如果 `C.f()` 调用了 `B.f()`，那么 `B.f()` 调用 `A.f()` 是很合理的。当用到覆盖 `method` 时，像这样的链接方式是相当常见的，同时它也是一个非常好的方式，可用来增加 `method` 的功能但不需将整个 `method` 完全换掉。我们在稍早就看过使用这种技巧的 `finalize()` 范例，该 `method` 调用了 `super.finalize()` 来执行它的超类的终止函数 `method`。

不要将使用 `super` 来调用被覆盖 `method` 与在构造函数中使用 `super()` 来调用超类的构造函数相混淆了，虽然两者都使用了相同的关键字，但却是完全不同的语法。在实际中，你可以使用 `super` 在覆盖类里的任何地方来调用被覆盖的 `method`，但你只可以在该构造函数的第一条语句中使用 `super()` 来调用超类的构造函数。

另外必须记得的是，`super` 只能被用在它所覆盖的 `method` 的类中来调用它所覆盖的 `method`。对于某个 `Ellipse` 类型的对象 `e`，你无法在程序中使用该对象（不管有没有 `super` 语法）来调用 `Circle` 类所定义的 `area()` `method`。

## 数据隐藏与封装

在本章一开始时，我们就介绍过类是“数据与方法的集合”。到目前为止，还有一个很重要的面向对象技术我们尚未讨论，那就是类的数据隐藏（`data hiding`），而且只能通



过 method 来引用这些数据。这样的技术通常叫做封装 (encapsulation)，因为它可将类的数据（包括内部 method）安全地封装在类这个“密封体” (capsule) 里，这样它能够让被信赖的用户（即类中的 method）来使用。

为什么要这样做呢？最重要的原因就是隐藏类的内部实现细节。如果你不想让其他的程序员接触到这些细节，那么你就可以安全地修改类的实现，而不用担心已经使用这个类的程序代码会因为你的修改而产生任何的副作用。

另一个使用封装的原因是为了要保护你的类，以避免碰上意外或故意的错误。一个类通常包含许多的字段，而它们彼此处于一致的状态之下。如果你允许程序员（包括你自己在内）直接操作这些字段，就可能会更改一个字段而没有更改其他重要的相关字段，因而让类产生了不一致的状态。如果换成必须调用一个 method 才能更改字段，而这种方法可以确保完成所有必要的操作以保持类的一致性。同样地，如果类定义了某些只提供内部使用的 method，则将这些 method 隐藏起来就可以避免用户调用它们。

关于封装的另一个想法是：当一个类的所有数据都隐藏起来时，类的 method 就可以定义这个类的对象唯一可能执行的运算。一旦 method 经过小心的测试与调试之后，你就可以预期它会正常运作。另一方面，如果该类的所有字段可以直接地被操作，那么调试的过程可能就会难以控制。

将类中的字段与 method 隐藏的理由还有：

- 若内部的字段与 method 可以让外部看见，会让类的 API 变得混乱。所以保持可见字段最少的状态，会让你的类显得更为简洁，同时让它更易于使用与理解。
- 如果类中的字段或 method 是可见的，你就必须以文件来加以说明。如果想要节省时间就把它给隐藏起来吧！

## 访问控制

类中的所有字段与 method 在该类的主体内都可以被使用。Java 定义了访问控制规则来限制类的外部对成员的访问。在本章的几个范例中，你已经看到了 public 修饰符被用在字段与方法的声明中。public、protected 与 private 都是访问控制修饰符 (access control modifier)，它们为字段与 method 设定了访问规则。

## 对包的访问

包可以被同一个包中的程序代码使用，至于包是否可以被其他包中的程序代码访问，则视该系统的包的发布方式而定。例如，当构成包的类文件被存储在一个目录中时，用户

必须对该目录以及该目录下的所有文件拥有读取的权利以便访问该包。包的访问控制并不是 Java 语言本身的一部分。访问控制通常是在类与类成员的层次完成的。

## 对类的访问

在默认的情况下，最顶层的类只能在它们所被定义的包中被使用。然而，如果最顶层的类被声明为 `public`，它就可以在任何的地方（或该包所能被访问到的任何地方）被使用。我们之所以要对最顶层的类做这样的限制的理由是，就像在本章的前面所看到的，类可以被定义在其他类的成员，因为这些内部类（inner class）是某个类的成员，所以它们都必须遵守成员的访问控制规则。

## 对成员的访问

类成员一定可以在该类的主体内被访问到。默认情况下，成员也可以在定义该类的包中被访问，这意味着在同一个包中的所有类应该互相信任彼此内部的实现细节。这个默认的访问级别通常都被称作包访问（package access），它只是四种访问级别中的一种而已，其他的三种分别分别用 `public`、`protected` 与 `private` 修饰符来定义。以下是使用这些修饰符的程序代码范例：

```
public class Laundromat {           // 所有人都可以使用这个类
    private Laundry[] dirty;        // 他们不能使用这个内部字段，
    public void wash() { ... }      // 但可以使用这些公开的 method
    public void dry() { ... }       // 来操作这些内部字段
    protected int temperature;     // 子类可能想要搞乱这个字段
}
```

以下的访问规则适用于类成员：

- 如果类中的成员被声明为 `public`，则表示只要是在包含该成员的类可以被访问的任何地方，该成员就可以被访问。这是访问控制中限制最少的类型。
- 如果类中的成员被声明为 `private`，则表示除了该类里的成员之外，所有外来的其他类成员都不能访问它。这是访问控制中限制最多的类型。
- 如果类中的成员被声明为 `protected`，则表示它可以被该包中的所有类访问（与默认的包访问一样），同时也可以被该类的所有子类的主体访问，不管该子类是否是在同一个包中。此访问控制比 `public` 的访问限制要多，但又比包的访问限制要少。
- 如果类中的成员都没有使用上述这些修饰符时，则它只拥有默认的包访问级别：它可以被同一个包中的所有类的程序代码所访问，但对于包外的类则是不可访问的。

`protected` 访问控制在使用上需要格外的小心。现在假设类 A 声明了一个 `protected` 类型的字段 `x`，且被定义于其他包中（这点很重要）的类 B 所扩展。类 B 继承了类 A 的

protected 字段 x，同时它的程序代码可以通过 B 的实例或以 B 的实例来访问字段。然而，这并不代表类 B 的程序代码可以随意的读取 A 实例的 protected 字段。如果有一个对象是 A 的一个实例，但并不是 B 的一个实例，那么很明显该字段并没有被 B 所继承，而且类 B 的程序代码是不可以读取它们的。

## 访问控制与继承

根据Java所定的规则，子类会继承它的父类中所有可以访问的实例字段与实例 method。如果子类被定义在与超类相同的包中，则它会继承所有非 private 的实例字段与 method。如果子类被定义在另一个包中，则它会继承所有 protected 与 public 实例字段与 method，private 类型的字段与 method 永远都不会被继承，类字段和类 method 也不会被继承。最后，构造函数也不会被继承，我们在本章的前面已经介绍过了，它们是链接的。

子类无法继承它的超类的不可访问的字段与 method 是一件相当令人困扰的事，这似乎是在暗示着当你创建子类的实例时，超类所定义的 private 字段都没有被分配内存。事实上，每一个子类的实例都包含一个完整的超类实例，包括所有的不可访问的字段与 method。这只是一个术语上的问题而已，因为不可访问的字段不能够被子类所使用，所以我们说它们并没有被继承。在本章的前面我们也曾经说过，类中的成员总是可以被该类的主体所访问到，如果这句话用于类中的所有成员并包括被继承的成员的话，我们就必须将“被继承成员”定义为可被访问的成员。如果你不计较这个定义，可以改用以下方式来思考：

- 一个类继承了它的超类的所有实例字段与实例 method（但不包括构造函数）。
- 类的主体总是可以被它自己所声明的字段与 method 访问，它也可以访问继承自其超类且可访问的字段与成员。

## 成员访问总结

表 3-1 总结了成员访问的规则。

表 3-1：类成员的访问

可供访问的对象	成员可见性			
	public	protected	package	private
定义类	是	是	是	是
相同包中的类	是	是	是	否
不同包中的子类	是	是	否	否
不同包中的非子类	是	否	否	否



这里有些使用显式 (visibility) 修饰符的简单规则:

- 只有构成类中的公共API部分的method和常量才会使用public。一些非常重要或是常常使用的字段也可以使用public, 但实际上字段通常是定义为非public, 然后再使用public类型的访问method将它们封装起来。
- 大多数的程序员在使用类时, 不太会使用到protected, 但是在另外一个包创建子类可能会使用到的字段与method时就会使用到protected。请注意, protected成员在技术上是该类的输出API的一部分, 它们必须使用文件加以说明。同时, 不可以没有修正依赖于它的其他程序代码的情况下就任意修改它。
- 如果你想要在该包之外隐藏字段和method, 但是在相同包的类中可以访问它们, 你可以使用默认的包可见性。而你只有在使用package指令将一群会互相合作的类分组到同一包中之后, 你才能使用此机制。
- 对于只用在类中且在其他地方应该隐藏的字段和method, 就使用private。

如果你并不确定到底该使用protected、package或private中的哪一个时, 刚开始时把成员访问权限制得严格一些可能会比较好。你在任何时候都可以依自己的需要来放宽这些限制。反之, 则是不智之举, 因为访问的限制程度是不具有向下兼容性的, 你先前所写的程序代码可能因此而无法正常运行。

## 数据访问 method

在Circle的例子中, 我们将圆的半径声明为一个public字段。在Circle类中让此字段可被公共访问才是有意义的, 这是一个很简单的类, 它与它的字段之间是互相独立的。另一方面, 我们目前对此类的实现允许Circle对象拥有一个负值的半径, 而半径为负值的圆实际上是不存在的。然而, 只要半径是存储于一个public类型的字段中, 则任何程序员就可以给该字段赋予他们所需要的值, 而不管该数值是多么的不合理。唯一的解决方法是限制程序员对该字段的直接访问, 并且定义一个public方法来对该字段提供非直接的访问。对该字段提供public方法以执行读写的操作并不等同于将该字段声明为public, 重大的不同点是在于方法可以执行差错校验。

例3-4显示了我们重新实现Circle以避免产生半径为负值的Circle对象。此版本的Circle声明了protected类型的字段r以及定义了名为getRadius()与setRadius()的两个访问method来读取与写入字段值, 并确保不会有负值的状况发生。因为字段r是protected类型的, 所以它可以被子类直接地 (或者说更有效率地) 访问。

例 3-4: 使用了数据隐藏与封装的 Circle 类

```
package shapes;           // 指定类的包
```

```
public class Circle {    // 此类仍是public类型的
    // 这是一个相当有用的常量，所以我们仍将它定义为public类型的
    public static final double PI = 3.14159;

    protected double r;    // 半径被隐藏了，但可被子类看见

    // 一个用来限制圆半径值的方法
    // 此为细节，对子类来说也许是有用的
    protected void checkRadius(double radius) {
        if (radius < 0.0)
            throw new IllegalArgumentException("radius may not be negative.");
    }

    // 构造函数method
    public Circle(double r) {
        checkRadius(r);
        this.r = r;
    }

    // 公用数据访问method
    public double getRadius() { return r; }
    public void setRadius(double r) {
        checkRadius(r);
        this.r = r;
    }

    // 用来操作实例字段的method
    public double area() { return PI * r * r; }
    public double circumference() { return 2 * PI * r; }
}
```

我们已经将Circle类定义在一个名为shapes的包中。因为r是protected类型的，所以任何在shapes包内的类都可以直接访问该字段，且可以依他们的需要来设定它。这里假设在shapes包里的所有类的作者皆为同一个人或是一个关系相当密切的工作小组，所有的类皆彼此相互信任，且不会滥用它们所拥有的访问权限来影响彼此的实现细节。

最后，用来限制半径值不得为负值的程序代码是一个名为checkRadius()的protected method。虽然Circle类的用户无法调用这个method，但该类的子类可以调用它，甚至当它们想要改变半径的限制时还可以覆盖它。

请特别注意，例3-4中的getRadius()与setRadius() method。在Java中对数据访问method几乎都是以“get”与“set”为开头。如果被访问的字段类型为boolean，则get()方法可以换做以“is”开头。例如，对一个名为readable的boolean类型的字段的访问方法来说，我们都会将它命名为isReadable()而不是getReadable()。在JavaBeans组件模型（将会在第七章做说明）的程序设计惯例中，拥有一个或一个以上以“get”、“is”或“set”开头的隐含字段被称作属性（property）。研究一个复杂类有一

个很有趣的方法，就是去看它所定义的属性。属性在 AWT 与 Swing API 里是很常见的，它们在《JAVA Foundation Class in a Nutshell》(O'Reilly) 一书中有深入的讨论。

## 抽象类与方法

在例 3-4 中，我们将 Circle 类声明为 shapes 包的一部分。假设我们要实现一些形状类：Rectangle、Square、Ellipse、Triangle 等。我们可以将两个基本的 area() 与 circumference() method 定义给这些类。现在，要让形状的数组变得容易使用，如果我们所拥有的形状类都有一个共同的超类 Shape 时，这将会是很有帮助的。如果我们使用这种方法来构造我们的类层次结构，则对于每一个形状对象，不管它所真正代表的形状是什么，都可以被指定为 Shape 类型的变量、字段或数组元素。我们希望 Shape 类可以封装所有形状类的共同特征（如 area() 与 circumference() 方法）。但通用的 Shape 类并不会代表任何实际的形状，所以它并不会定义有用的 method 的实现。Java 使用抽象（abstract）method 来处理这样的情况。

Java 让我们将 method 声明为 abstract 来定义一个不用实现的 method。abstract method 不会有程序主体，它在签名的定义后面接着一个分号（注 8）。以下是关于 abstract method 与包含它们的 abstract 类的一些规则：

- 任何包含 abstract method 的类都会自动地将自己变成 abstract 类，并且它也必须被声明为 abstract。
- abstract 类不能被实例化。
- abstract 类的子类只有在它覆盖了它的超类的 abstract 方法并把它们全部实现（也就是方法的主体）的情况下才可以被实例化。这样的类常被称作具体（concrete）子类，用来强调它并不是 abstract。
- 如果 abstract 类的子类没有把它所继承的 abstract method 实现出来，那么它还是一个 abstract 类。
- static、private 与 final 方法都不得为 abstract，因为这些类型的方法都不能被子类所覆盖。同样地，final 类也不能含有任何 abstract method。
- 一个类可以被声明为 abstract，即使它并不一定拥有任何的 abstract method。

注 8：Java 里的 abstract method 有时候和 C++ 里的纯虚函数是一样的（也就是，被声明为 = 0 的虚函数）。在 C++ 里，包含了纯虚函数的类被称为抽象类，而且不可以被实例化。这对于包含 abstract method 的 Java 类也成立。



声明像这样的 `abstract` 类表示它的实现并不完全,而且必须交由子类来完成。像这样的类也无法被实例化。

`abstract method` 有一个相当重要的特点,如果我们所定义的 `Shape` 类有 `abstract area()` 与 `circumference()` 两种方法,则任何 `Shape` 的子类都必须定义这两个方法的实现,这些方法才可以被实例化。换句话说,每一个 `Shape` 对象都必须保证包括定义的这两个方法的具体实现。例 3-5 说明了上面的这条理论,它定义了一个 `abstract Shape` 类与两个具体的子类。

例 3-5: `abstract` 类与具体子类

```
public abstract class Shape {
    public abstract double area();           // 抽象方法,请注意
    public abstract double circumference();  // 以分号来取代主体
}

class Circle extends Shape {
    public static final double PI = 3.14159265358979323846;
    protected double r;                    // 实例数据
    public Circle(double r) { this.r = r; } // 构造函数
    public double getRadius() { return r; } // 访问器
    public double area() { return PI*r*r; }  // 抽象method的实现
    public double circumference() { return 2*PI*r; }
}

class Rectangle extends Shape {
    protected double w, h;                  // 实例数据
    public Rectangle(double w, double h) {   // 构造函数
        this.w = w; this.h = h;
    }
    public double getWidth() { return w; }   // 访问器 method
    public double getHeight() { return h; }  // 另一个访问器
    public double area() { return w*h; }     // 抽象method的实现
    public double circumference() { return 2*(w + h); }
}
```

每一个在 `Shape` 里的 `abstract method` 在它的小括号后面都会有一个分号。这里并没有大括号,而且也没有定义 `method` 主体。使用定义于例 3-5 里的类,我们可以编写如下的程序代码:

```
Shape[] shapes = new Shape[3];           // 创建一个数组来存放 shape
shapes[0] = new Circle(2.0);              // 为数组赋
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);

double total_area = 0;
for(int i = 0; i < shapes.length; i++)
    total_area += shapes[i].area();       // 计算每种形状的面积
```

在这里有很重要的两点要注意:

- Shape的子类可以被赋值给Shape的数组元素而不需使用强制转换。这是另一个关于放大引用类型转换的例子（第二章所讨论的）。
- 任何的Shape对象都可以调用area()与circumference() method, 即使Shape类没有定义这些method的主体。当你这么做时, 被调用的method会通过动态method查找而被找到, 因此将会使用Circle所定义的method来计算圆的面积, 而长方形的面积则会使用Rectangle所定义的method来计算。

## java.lang.Object 的重要 method

就如我们所知道的, 所有的类都是直接地或间接地扩展自 java.lang.Object。这个类定义了几个重要的 method, 在你所编写的每一个类里, 你必须考虑覆盖 (overriding) 那几个 method。例3-6显示了一个类覆盖了这些 method。接在范例后的一小段文字说明了每一个方法的默认实现方法以及解释了为什么你会想要覆盖它。

在例3-6里的一些语法对你来说可能会有些陌生。该范例使用了两个Java 5.0的功能。第一, 它实现了参数化 (parameterized)、泛型 (generic) 及此版本的Comparable接口。第二, 该范例使用了@Override annotation来强调 (并有编译器来验证) 该 method 覆盖了Object。参数化类型 (parameterized type) 与 annotation 会在第四章做介绍。

例 3-6: 覆盖了重要 Object method 的类

```
// 这个类代表一个具有固定位置与半径的圆
public class Circle implements Comparable<Circle> {
    // 这些字段保存圆心与半径
    // private 是针对数据封装, 而 final 则是针对不变性
    private final int x, y, r;

    // 基本的构造函数: 初始化字段并指定值
    public Circle(int x, int y, int r) {
        if (r < 0) throw new IllegalArgumentException("negative radius");
        this.x = x; this.y = y; this.r = r;
    }

    // 这是一个“复制构造函数”—— 有用的 clone() 替代方案
    public Circle(Circle original) {
        x = original.x;    // 从 original 复制字段
        y = original.y;
        r = original.r;
    }

    // private 字段的公共访问 method
    // 这些都是数据封装的一部分
    public int getX() { return x; }
    public int getY() { return y; }
    public int getR() { return r; }
```

```

// 返回一个字符串表达式
@Override public String toString() {
    return String.format("center=(%d,%d); radius=%d", x, y, r);
}

// 对另一个对象做相等性测试
@Override public boolean equals(Object o) {
    if (o == this) return true;           // 是否为相同的调用?
    if (!(o instanceof Circle)) return false; // 类型正确且非null?
    Circle that = (Circle) o;             // 强制转换为我们所要的类型
    if (this.x == that.x && this.y == that.y && this.r == that.r)
        return true;                     // 如果所有的字段都相符
    else
        return false;                   // 如果字段不同
}

// 哈希码能让对象被用于哈希表内
// 相等对象必须具有相等的哈希码, 不相等的对象也能有相等的哈希码,
// 但我们会尝试着避免这样的情形发生
// 我们必须覆盖这个method, 因为我们也覆盖了 equals()
@Override public int hashCode() {
    int result = 17;                     // 这是来自于由Joshua Bloch所著的
    result = 37*result + x;              // 《Effective Java》一书的哈希码算法
    result = 37*result + y;
    result = 37*result + r;
    return result;
}

// 这个method由Comparable接口所定义
// 将this Circle与that Circle作比较: 如果this < that 则返回值 < 0;
// 如果this == that 则返回0; 如果this > that 则返回值 > 0
// Circle是有顺序性的, 从上而下, 从左而右, 依半径值来决定
public int compareTo(Circle that) {
    long result = that.y - this.y; // 较小的圆有较大的y值
    if (result == 0) result = this.x - that.x; // 如果相同, 则从左到右比较
    if (result == 0) result = this.r - that.r; // 如果相同, 则比较半径

    // 对于减法运算我们必须使用long值, 因为对于int来说,
    // 很大的正值与很大的负值都可能会溢位 (overflow) 一个int。
    // 但我们不能返回 long 类型的值, 所以返回它的符号来作为一个int
    return Long.signum(result); // Java 5.0中新加的功能
}
}

```

## toString()

toString() method 的主要的目的是返回对象的文字表达式。此 method 会在字符串连接时或通过像 System.out.println() 这样的 method 被自动调用。给予对象一个文字表达式, 在调试或记录输出时会非常的有帮助, 同时这个制作相当好的 toString() method 甚至对于我们的像是产生报表这样的工作很有帮助。



此版本的 `toString()` 继承自 `Object`，同时会返回一个字符串，该字符串包含了该对象的类的名称，而且和该对象的 `hashCode()` method 所表示的十六进制值一样（稍后会做说明）。这个默认的实现方法为该对象提供了基本的类型与该对象本身的一些信息，但并不常被使用到。例 3-6 里的 `toString()` method 以一串易懂的字符串表达式将 `Circle` 类内的每一个字段值返回。

## `equals()`

`==` 运算符测试两个引用是否是引用相同的对象。如果你要测试两个不同的对象是否彼此相等时，就必须使用 `equals()` method。每一个类可以通过覆盖 `equals()` method 来定义他们自己对相等的看法。`Object.equals()` method 就只是使用了 `==` 运算符而已，只有在他们的是相同对象的情况下，这个默认的 method 才将会两个对象视为相等。

如果它们的两个不同的 `Circle` 对象的字段都是相等的，则例 3-6 的 `equals()` method 会将它们视为是相等的。请注意，该 method 首先会使用 `==` 来做快速相等性测试，然后使用 `instanceof` 来检查其他对象的类型：`Circle` 只能和另一个 `Circle` 相等，如果该对象是不可接受的类，则 `equals()` method 会抛出 `ClassCastException`。`instanceof` 测试也会排除 `null` 自变量：如果在 `instanceof` 左边的操作数为 `null`，则 `instanceof` 计算出来的值都是 `false`。

## `hashCode()`

当你覆盖了 `equals()` method 时，你也必须要将 `hashCode()` method 覆盖。`hashCode()` method 会返回一个 `integer` 值给哈希表数据结构使用。如果两个对象在 `equals()` method 的结果为相同的情况之下，则这两个对象就会有相同的哈希码。这是非常重要的（对哈希表做有效的运算），但对于有不同的哈希码的不相等的对象来说就不需要了，且不相等的对象不太可能会去共享一个哈希码。第二个准则可以让 `hashCode()` method 适当地调用较复杂的运算或位运算（bit-manipulation）。

`Object.hashCode()` method 配合 `Object.equals()` method 使用，会返回以对象相同性为基础的哈希码。它不是以对象相等性为基础（如果你需要以相同性为基础（identity-based）的哈希码，可以通过静态 method `System.identityHashCode()` 来访问 `Object.hashCode()` 的功能）。当你在覆盖 `equals()` 时，也一定要覆盖 `hashCode()` 来保证相等的对象具有相等的哈希码。因为在例 3-6 里的 `equals()` method 是以三个字段的值作为对象相等性的基础，所以 `hashCode()` method 也同样会以这三个字段值来计算它的哈希码。从程序代码来看，有件事很清楚，那就是如果两个 `Circle` 对象具有相同的字段值，它们就会有相同的哈希码。

请注意, 例 3-6 里的 `hashCode()` method 并不是将三个字段值给加起来并返回其总和。这样的实现过程是合法但却是没有效率的, 因为两个圆可能半径相同但 X 与 Y 坐标却相反, 可是它们还是有相同的哈希码。重复的乘法与加法运算将哈希码的范围给“扩展开”了, 同时减少了两个不同的 `Circle` 对象具有相同哈希码的可能性。Joshua Block (Addison Wesley) 所著的《Effective Java Programming Guide》一书里介绍了构造像这个 `hashCode()` method 一样有效率的 method 的方式。

## Comparable.compareTo()

例 3-6 里有一个 `compareTo()` method, 这个 method 是由 `java.lang.Comparable` 接口所定义的, 而不是由对象所定义的 (它使用了 Java 5.0 的泛型特点, 同时它也实现了接口的参数化 (parameterized): `Comparable<Circle>`, 但在尚未介绍到第四章之前, 我们可以先不管)。Comparable 与 `compareTo()` method 的用途就是允许类的实例之间使用 `<`、`<=`、`>` 与 `>=` 运算符来互相比。如果有一个类实现了 `Comparable`, 我们可以说一个实例小于、大于或等于其他的实例。Comparable 类的实例可以被排序。

因为 `compareTo()` 是被接口所定义的, 所以 `Object` 类没有提供任何的默认实现方法。由每一个独立的类来决定它的实例是否或如何被排序, 并包含了实现此顺序的 `compareTo()` method。例 3-6 定义了它的顺序性, 它将几个 `Circle` 对象拿来作比较。`Circle` 首先会以由上而下的顺序来作比较: 具有较大 Y 坐标的圆小于具有较小 Y 坐标的圆。如果两个圆具有相同的 Y 坐标, 它们就会依从左到右的顺序来作比较: X 坐标较小的圆小于 X 坐标较大的圆。最后, 如果两个圆有相同的 X 与 Y 坐标的话, 就要比较它们的半径了: 拥有较小半径的圆当然就比较小。在这样的比较过程中要注意的是, 只有在三个字段都相等的情况下, 两个圆才会相等。这意味着 `compareTo()` 所定义的顺序性与 `equals()` 所定义的相等性是一致的。

`compareTo()` method 所返回的 `int` 值需要更进一步地加以说明。如果 `this` 对象小于传递进来的对象时, `compareTo()` 会返回负值; 如果两个对象相等时, 则会返回 0; 如果 `this` 对象大于传递进来的对象时, 则 `compareTo()` 会返回正值。

## clone()

`Object` 定义了一个名为 `clone()` 的 method, 它的主要目的是返回一个与当前对象具有完全相同的字段集的对象。此 method 之所以不常被使用的两个原因为: 第一, 只有在类实现了 `java.lang.Cloneable` 接口时才可以使用它, `Cloneable` 没有定义任何的 method, 所以它的实现就取决于类签名的 `implements` 子句; 另一个原因是, 它被声明为 `protected` (请看先前的“数据隐藏与封装”章节), 也就是说, `Object` 的子

类可以调用以及覆盖 `Object.clone()`，但其他的程序代码就不可以调用它。所以，如果你希望你的对象具有可复制性，就必须实现 `Cloneable` 并覆盖 `clone()` method，让它成为 `public` 类型的。

例 3-6 里的 `Circle` 类并没有实现 `Cloneable`，但它提供了复制构造函数（copy constructor）来对 `Circle` 对象做复制的操作。

```
Circle original = new Circle(1, 2, 3); // 一般的构造函数
Circle copy = new Circle(original);    // 复制构造函数
```

要正确地实现 `clone()` 可能会很困难，而提供一个复制构造函数通常是一个较容易且安全的方式。为了让 `Circle` 类具有可复制性，你可以在 `implements` 子句里加入 `Cloneable` 并将以下的 method 加入类主体中：

```
@Override public Object clone() {
    try { return super.clone(); }
    catch(CloneNotSupportedException e) { throw new AssertionError(e); }
}
```

在 Joshun Bloch 的《Effective Java Programming Guide》一书中对于 `clone()` 的所有情况以及 `Cloneable` 有更详细的介绍。

## 接口

跟类一样，接口（interface）定义了一个新的引用类型。然而，和类不一样的是，接口不会为它们所定义的类型提供实现。就如同字面上所提示的，一个接口仅指定一个 API；该接口的所有方法都是抽象的（abstract）而且也没有程序主体。如果要直接实例化（instantiate）一个接口并创建属于该接口类型的成员是不可能的。类必须实现接口来提供必需的 method 主体。该类的任何实例都应该是类所定义的类型以及接口所定义的类型成员。接口提供了有限但非常强大的多重继承（multiple inheritance）（注 9）。Java 里的类只能从单一的超类继承成员，但它们可以实现任意数量的接口。通过实现相同的接口，未共享相同类或超类的对象仍然可以是相同类型的成员。

### 定义接口

接口的定义和类的定义非常相似，但在接口里所有的 method 都是抽象的，而关键字 `class` 被换成 `interface`。例如，以下就是一个名为 `Centered` 的接口的程序代码。就

注 9： C++ 支持多重继承，但一个类可以拥有多个超类的功能为这个语言增加了许多复杂性。



像本章稍早所定义的Shape类,如果它想让它的坐标的中心能被设置或查找时,就应该实现这个接口:

```
public interface Centered {  
    void setCenter(double x, double y);  
    double getCenterX();  
    double getCenterY();  
}
```

以下是适用于此接口的一些限制:

- 接口不包含任何的实现。接口的所有method都是abstract,而且必须有一个分号来代替method主体。习惯上,abstract修饰符是被允许的,但通常会省略。因为静态method不可以是抽象,而接口的method也不可以被声明为static。
- 接口定义了一个公共API。接口的所有成员都为public类型,且通常这个非必要的public修饰符会被省略。在接口里不可以定义protected或private类型的method。
- 接口不可以定义任何的实例字段。字段是实例的细节,而接口纯粹只是一个声明,不包含任何的实例。在接口定义里唯一被允许的字段是声明为static和final的常量。
- 接口不可以被实例化,所以没有定义构造函数。
- 接口可以包含嵌套类型。任何这样的类型都是public与static类型的。请看本章稍后的“嵌套类型”一节。

## 扩展接口

接口可以扩展其他的接口,而且就像类定义一样,接口定义可以包含extends子句。当一个接口扩展另一个接口时,它会继承它的父接口(superinterface)的所有抽象method与常量,而且可以定义新的抽象method与常量。然而,和类不一样的是,接口定义的extends子句可以包含一个以上的父接口。例如,以下就是一些扩展其他接口的接口:

```
public interface Positionable extends Centered {  
    void setUpperRightCorner(double x, double y);  
    double getUpperRightX();  
    double getUpperRightY();  
}  
public interface Transformable extends Scalable, Translatable, Rotatable {}  
public interface SuperShape extends Positionable, Transformable {}
```

扩展了多个接口的接口会继承来自这些接口的所有抽象method与常量,而且可以定义它自己的抽象method与常量。实现了这种接口的类也必须实现由接口定义的抽象method,以及所有从父接口继承来的抽象method。

## 实现接口

就像类使用了 `extends` 来声明它的超类一样, `implements` 也可被用来命名它所支持的一个或多个接口。 `implements` 是 Java 的关键字, 它出现在类声明中的 `extends` 子句后面, `implements` 后面必须接着该类所实现的接口名称。若有多个接口时, 必须使用逗号加以分隔。

当类在它的 `implements` 子句里声明接口时, 就可以说是它为该接口的每一个方法提供了实现 (如主体)。如果类实现了接口但没有为该接口的所有 `method` 提供实现, 它就会从接口继承未实现的抽象 `method`, 而且它本身必须要被声明为 `abstract`。如果类实现了多个接口, 就必须实现它所实现的各个接口里的每一个 `method` (或把自己声明为 `abstract`)。

以下程序代码说明了我们该如何定义一个 `CenteredRectangle` 类, 它扩展了本章先前的 `Rectangle` 类, 并实现了我们定义的 `Centered` 接口。

```
public class CenteredRectangle extends Rectangle implements Centered {
    // 新的实例字段
    private double cx, cy;

    // 一个构造函数
    public CenteredRectangle(double cx, double cy, double w, double h) {
        super(w, h);
        this.cx = cx;
        this.cy = cy;
    }

    // 我们继承了 Rectangle 类的所有 method,
    // 但必须提供所有 Centered method 的实现
    public void setCenter(double x, double y) { cx = x; cy = y; }
    public double getCenterX() { return cx; }
    public double getCenterY() { return cy; }
}
```

假设我们实现了 `CenteredCircle` 与 `CenteredSquare`, 就像我们实现了 `CenteredRectangle` 类一样。因为每一个类都扩展了 `Shape`, 这些类的实例都可以被视为 `Shape` 类的实例。因为每一个类都实现了 `Centered` 接口, 所以这些类的实例也可以被视为是该类型的实例。以下的程序代码将说明对象如何既可以是类的成员也可以是接口的成员:

```
Shape[] shapes = new Shape[3];           // 创建一个数组来存储这些形状

// 创建一些 Centered 类型的形状并将它们存储于 Shape[] 中
// 不需要使用到强制转换, 这些都是放大转换
shapes[0] = new CenteredCircle(1.0, 1.0, 1.0);
shapes[1] = new CenteredSquare(2.5, 2, 3);
shapes[2] = new CenteredRectangle(2.3, 4.5, 3, 4);
```

```
// 计算这些形状的平均面积以及到坐标原点的平均距离
double totalArea = 0;
double totalDistance = 0;
for(int i = 0; i < shapes.length; i++) {
    totalArea += shapes[i].area(); // 计算形状的面积
    if (shapes[i] instanceof Centered) { // 这是一个Centered类型的形状
        // 请注意从Shape到Centered的强制转换是必要的
        // (然而, 从CenteredSquare到Centered则不需要做强制转换)
        Centered c = (Centered) shapes[i]; // 将它赋值给一个Centered变量
        double cx = c.getCenterX(); // 获得圆心的坐标
        double cy = c.getCenterY(); // 计算与原点的距离
        totalDistance += Math.sqrt(cx*cx + cy*cy);
    }
}
System.out.println("Average area: " + totalArea/shapes.length);
System.out.println("Average distance: " + totalDistance/shapes.length);
```

这个范例说明了接口就像类一样也是Java的数据类型。当类实现接口时, 类的实例可以被赋值给接口类型的变量。请不要将这个范例解释为在调用setCenter() method之前必须将CenteredRectangle对象赋值给Centered变量, 或者是在你调用area() method之前要把CenteredRectangle对象赋值给Shape变量。CenteredRectangle定义了setCenter()并从它的超类Rectangle继承了area(), 所以你一定可以调用到这些method。

## 实现多个接口

假设我们希望形状对象不只可以依据它们的中心来加以定位, 而且能够根据它们右上角的坐标来定位。假设我们也想让形状可以被放大或缩小。应记住虽然一个类只可以扩展一个超类, 但它却可以实现任意数量的接口。假设我们已经定义了正确的UpperRightCornered与Scalable接口, 我们可以将这个类声明如下:

```
public class SuperDuperSquare extends Shape
    implements Centered, UpperRightCornered, Scalable {
    // 这里省略了类成员
}
```

当类实现多个接口时, 这表示它必须实现这些接口里的所有抽象method。

## 接口与抽象类

当在定义一个抽象类型(如Shape)时, 你预期会有很多子类型(subtype, 如Circle、Rectangle、Square), 所以常要在接口与抽象类之间作选择。因为它们有相似的特点, 所以要使用哪一个常是不确定的。

接口之所以有用是因为所有的类都可以实现它, 即使该类扩展了某个完全不相关的超类。



但接口纯粹只是一个API规范且不包含任何的实现。如果该接口有很多method,不断地实现这些method会是相当繁琐的一件事,尤其是在每个实现类中的实现都重复的情况下。

一个抽象类并不需要完全是抽象的,它可以拥有部分的实现,而子类也可以从中获得一些益处。在某些状况下,许多的子类可以依赖于抽象类所提供的默认method实现。但一个类扩展了某个抽象类后就不可以再扩展其他的类了,这样会在某些情况之下造成设计上的困扰。

接口与抽象类之间的另一个很重要的差异是兼容性的问题。如果你定义一个接口作为公共API的一部分,之后又为该接口加入了一个新的method,就会将使用前一版的接口来实现的类破坏掉。然而,如果你使用抽象类的话,就可以安全地在该类中加入非抽象的method,而不需要修改之前扩展该抽象类的类。

在某些情况之下,可以确定接口或抽象类是正确的设计选择。但在其他的情况下,常见的设计模式是两种都使用。先将类型定义为完全抽象的接口,然后创建实现了接口的抽象类,并提供一些有用的默认实现好让子类加以利用。例如:

```
// 这是一个基本的接口, 它代表了矩形所必须包含的组件,
// 任何想实现 RectangularShape 的类都可以实现这些 method
public interface RectangularShape {
    void setSize(double width, double height);
    void setPosition(double x, double y);
    void translate(double dx, double dy);
    double area();
    boolean isInside();
}

// 此为该接口的部分实现, 许多的实现都可以将它当成雏形
public abstract class AbstractRectangularShape implements RectangularShape
{
    // 形状的位置与大小
    protected double x, y, w, h;

    // 某些接口方法的默认实现
    public void setSize(double width, double height) { w = width; h = height; }
    public void setPosition(double x, double y) { this.x = x; this.y = y; }
    public void translate (double dx, double dy) { x += dx; y += dy; }
}
```

## 标志接口

有时候定义完全空的接口也是很有用的。只要将该接口的名称放在implements子句中而不必实现任何的method,类就可以实现这样的接口。在这种情况下,该类的任何实例

都变成了接口的合法实例。Java 程序代码会通过 instanceof 运算符来检查对象是否为接口的实例，所以这个技术对于提供更多有关对象的信息是很有用的。

java.io.Serializable 就是这种标志接口。类实现 Serializable 接口来告诉 ObjectOutputStream，它的实例可以被安全地排序。java.util.RandomAccess 是另一个例子：java.util.List 实现提供了这个接口来声明它们对列表中的元素提供了快速的随机访问。例如，ArrayList 实现了 RandomAccess，而 LinkedList 则没有。对于关心随机访问操作性能的算法，可以用以下方式测试 RandomAccess：

```
// 在对一个长列表内的元素做排序之前，
// 我们要先确定该列表是否允许做快速随机访问。
// 如果不允许，则在排序之前它会立刻为该列表制作一个随机访问的副本。
// 请注意，当使用 java.util.Collection.sort() 时，就不需要这么做了
List l = ...; // 我们随意给的某个列表
if (l.size() > 2 && !(l instanceof RandomAccess)) l = new ArrayList(l);
```

## 接口与常量

就像之前所说的，常量可以出现在接口的定义中。任何实现一个该接口的类继承了它所定义的常量并且可以直接使用它们，就好像是该类自己定义的一样。重要的是，常量的前面不必加上接口的名称或提供任何种类的常量实现。

当一组常量被多个类使用时，较吸引人的做法是在接口里定义那些常量一次，然后让所有需要那些常量的类来实现接口。例如，当客户端与服务器类实现了一个网络协议，其细节（例如联机和监听所使用的端口号）会被放在一组符号常量中。以一个具体的例子来看，考虑 java.io.ObjectStreamConstants 接口，它为对象序列协议定义了常量，并且是由 ObjectInputStream 和 ObjectOutputStream 两个对象来实现。

继承来自于接口所定义的常量的最主要的好处，是可以省去输入的动作：你不需要为接口所定义的常量指定类型。尽管它使用了 ObjectStreamConstants，但不建议使用这种方式。常量的使用是实现的细节，将它声明于类签名的 implements 子句中是不适当的。

较好的做法是在类内部定义常量，同时通过输入完整的类名与常量名来使用这些常量。在 Java 5.0 及以上的版本中，你可以用 import static 声明从它们的定义类输入常量以省略输入的动作。细节请参阅第二章的“包与 Java 命名空间”。

## 嵌套类型

到目前为止，我们所看到的类、接口与枚举类型（enumerated type）都被定义为最顶层的类。这表示它们是包的直接成员，独立定义于其他的类型。然而，类型的定义也可以

被嵌套在其他类型里面。这些嵌套类型 (nested type)，就是所谓的“内部类 (inner class)”，它们是 Java 语法强大且精巧的特点。类型被其他类型嵌套的方式有四种：

### 静态成员类型

静态成员类型 (static member type) 是作为另一个类型的 static 成员被定义的类型。static method 被称作类 method，由此类推，我们可以称这类的嵌套类型为“类类型”，但这样的术语很明显地会令人感到困惑。静态成员类型的行为和一般的顶层类型非常相似，但该类型的名称是命名空间的一部分，而不是包的一部分。同样地，静态成员类型可以访问类的 static 成员。不管有没有 static 关键字，嵌套接口、枚举类型与 annotation 类型都是静态的。任何在接口或 annotation 里的嵌套类型也都是 static。静态成员类型可以被定义在顶层的类型中，或嵌套在其他静态成员类型的任意深度中。但是，静态成员类型不可被定义在任何种类的嵌套类型中。

### 非静态成员类

“非静态成员类型 (nonstatic member type)”只是个成员类型，它不可被声明为 static。由于接口、枚举类型与 annotation 都是 static，所以我们通常会使用“非静态成员类 (nonstatic member class)”来表示。非静态成员类可以被定义在其他的类或枚举类型里，而且它与实例 method 或字段都很相似。一个非静态成员类的实例总是与包含类的实例相关联，而且非静态成员类的程序代码可以访问它所在的类的所有字段与 method (不管是 static 还是非 static)。Java 语法有数个特性是为了使用非静态成员类的实例而设置的。

### 局部类

局部类 (local class) 是一个定义在 Java 代码块里的类。接口、枚举类型以及 annotation 类型不可以被定义为局部类。跟局部变量一样，局部类只有在该代码块内才是可见的。虽然局部类不是成员类，但是它们仍然可以在它们所处的类中被定义，因此它们有几个特性和成员类相同。此外，局部类可以访问任何的 final 局部变量或定义该类的代码块的范围中所能访问的参数。

### 匿名类

匿名类 (anonymous class) 是一称没有名称的局部类，它结合了类定义的语法与对象实例化的语法。局部类的定义是个 Java 语句，匿名类的定义 (与实例化) 是个 Java 表达式，因此它可以出现在更大的表达式中，例如 method 调用。接口、枚举类型与 annotation 类型不能够被定义为这种类型。

对嵌套类型的名称还尚未达成共识。“内部类 (inner class)”这个名称常被使用。然而，有时候“内部类”会被用来指非静态成员类、局部类或匿名类，但不包括静态成员类型。虽然用来描述嵌套类型的专有名词并不是那么的清楚，但它的语法却是相当的明确。



现在我们要开始更详细地介绍这四个嵌套类型了。以下的每一节都会介绍嵌套类型的特性、使用上的限制与这些类型所使用的特殊语法。紧接着这四个章节的就是说明嵌套类型的运作方式的实现注释。

## 静态成员类型

静态成员类型 (static member type) 和一般的顶层类型 (top-level type) 非常相似。然而为了方便, 它被嵌套在其他类型的命名空间里。例 3-7 显示了一个有用的接口, 它被定义为一个包含类的静态成员。此范例也显示了这个接口在类内部以及在类外部是如何被使用的, 请注意在外部类中所使用的层次结构名称。

例 3-7: 定义与使用一个静态成员接口

```
// 将堆栈用链表实现的类
public class LinkedStack {
    // 此静态成员接口定义了对象该如何被链接
    // static 关键字是非必要的: 所有的嵌套接口都是 static
    public static interface Linkable {
        public Linkable getNext();
        public void setNext(Linkable node);
    }

    // 链表的头节点是一个 Linkable 类型的对象
    Linkable head;

    // 省略 method 的主体
    public void push(Linkable node) { ... }
    public Object pop() { ... }
}

// 此类实现了静态成员接口
class LinkableInteger implements LinkedStack.Linkable {
    // 这是节点的数据与构造函数
    int i;
    public LinkableInteger(int i) { this.i = i; }

    // 这是实现接口所需要的数据与 method
    LinkedStack.Linkable next;
    public LinkedStack.Linkable getNext() { return next; }
    public void setNext(LinkedStack.Linkable node) { next = node; }
}
```

## 静态成员类型的特性

静态成员类型是被定义为它所在类型 (containing type) 的 static 成员, 任何的类型 (类、接口、枚举类型或 annotation 类型) 都可以被定义为任何其他类型的 static 成员。不管定义中有没有 static 关键字, 接口、枚举类型与 annotation 类型也都被定义为 static。

静态成员类型和类的其他静态成员很相似：静态字段与静态method。和类method一样，静态成员类型与该类的任何实例是不相关联的。然而，静态成员类型可以访问该类内部的所有静态成员（包括所有其他的静态成员类型）。静态成员类型可以使用任何其他的静态成员而不需要特别指出该类型的名称。

静态成员类型可以访问它所在类型中的所有静态成员，包括private成员。反之亦然，也就是包含类型的method也可以访问静态成员类型中的所有成员，包括private成员。静态成员类型同样也对任何其他静态成员类型中的所有成员拥有访问权，包括这些类型的private成员。

顶层类型在声明时使用或不使用public修饰符都可以，但却不可以使用private与protected修饰符。因为静态成员类型本身就是类型成员，所以在该类型内的其他成员可以使用任何的访问控制修饰符。这些修饰符对静态成员类型来说都具有相同的意义，就像它们对该类型的其他成员所做的一样。在例3-7里，Linkable接口被声明为public，因此它可以被任何想将数据存储在LinkedStack中的类来实现。应记住所有的接口成员（以及annotation类型）都是被定义为public，所以在接口或annotation类型里的静态成员类型不可以为protected或private。

### 静态成员类型的限制

静态成员类型的名称不能跟包含它的类的名称相同。此外，静态成员类型只可以在顶层类以及其他的静态成员类型中被定义，也就是说静态成员不能在成员类、局部类以及匿名类内被定义。

### 静态成员类型的语法

在包含类之外的程序代码中，静态成员类型的名称是结合了外部与内部的名称（如LinkedStack.Linkable），而你可以使用import指令来导入一个静态成员类型：

```
import pkg.LinkedList.Linkable; // 导入某个特定的嵌套类型
import pkg.LinkedList.*;        // 导入LinkedList的所有嵌套类型
```

在Java 5.0及之后的版本中，你也可以使用import static指令来导入一个静态成员类型。关于import与import static的细节，请参阅第二章中的“包与Java命名空间”。请注意，导入嵌套类型会遮盖住嵌套类型与包含类型的紧密关系，所以这种做法并不常见。

## 非静态成员类

非静态成员类是个被声明为不具有 `static` 关键字的包含类 (containing class) 或枚举类型的成员的类。如果静态成员类型与类字段或类 method 相似, 那么非静态成员类就与实例字段或实例 method 相似。例 3-8 显示了成员类如何被定义与使用。这个范例扩展了之前的 `LinkedStack` 例子, 它通过定义会返回 `java.util.Iterator` 接口的实现的 `iterator()` method 来允许对堆栈 (stack) 中元素的列举。这个接口的实现被定义为成员类。此范例在一些地方使用了 Java 5.0 的泛型 (generic type) 语法, 但这并不会对你造成妨碍 (generics 将在第四章中被介绍)。

例 3-8: 将 `iterator` 实现为成员类

```
import java.util.Iterator;

public class LinkedStack {
    // 我们的静态成员接口
    public interface Linkable {
        public Linkable getNext();
        public void setNext(Linkable node);
    }

    // 链表的头节点
    private Linkable head;

    // 这里省略了 method 的主体
    public void push(Linkable node) { ... }
    public Linkable pop() { ... }

    // 此 method 返回 LinkedStack 的 Iterator 对象
    public Iterator<Linkable> iterator() { return new LinkedIterator(); }

    // 这是 Iterator 接口的实现,
    // 它被定义为非静态成员类
    protected class LinkedIterator implements Iterator<Linkable> {
        Linkable current;
        // 构造函数使用包含类的专用 head 字段
        public LinkedIterator() { current = head; }
        // 以下的三个 method 都是由 Iterator 接口所定义
        public boolean hasNext() { return current != null; }
        public Linkable next() {
            if (current == null) throw new java.util.NoSuchElementException();
            Linkable value = current;
            current = current.getNext();
            return value;
        }
        public void remove() { throw new UnsupportedOperationException(); }
    }
}
```



请注意 `LinkedListIterator` 类被嵌套在 `LinkedList` 类中的方式。因为 `LinkedListIterator` 是只被用在 `LinkedList` 中的辅助类 (helper class)，所以在离使用到它的地方的不远处定义它，是相当正确的做法。

## 成员类的特性

就像实例字段与实例 method 一样，每一个非静态成员类的实例都与该类的实例相关。这表示成员类的程序代码可以访问包含类的所有实例字段与实例 method (与 `static` 成员一样)，包括所有声明为 `private` 类型的。

例 3-8 已经阐述过这个重要的特性了。以下为 `LinkedList.LinkedListIterator()` 构造函数：

```
public LinkedListIterator() { current = head; }
```

这一行代码将内部类的 `current` 字段设定为包含类的 `head` 字段值。正如此程序代码所示，即使 `head` 在包含类中是被声明为 `private` 类型的，也不会影响这条语句的正常执行。

一个非静态成员类，就像类中的其他成员一样，可以被指定为三种显式级别中的其中一种：`public`、`protected` 或 `private`。如果没有用这三种显式修饰符，就会使用到默认的显式级别。在例 3-8 里，`LinkedListIterator` 类被声明为 `protected`，所以使用 `LinkedList` 类的程序代码 (在不同包里) 将会无法访问它，但 `LinkedList` 子类里的所有类都可以访问它。

## 成员类的限制

成员类有三个相当重要的限制：

- 非静态成员类的名称不可以与它的包含类或包的名称相同。这是一条很重要的规则，而且它也不能被字段与 method 共享。
- 非静态成员类不可以包含任何 `static` 字段、method 或类型，除了同时使用了 `static` 与 `final` 的常量字段之外。静态成员是顶层结构，它不与任何特殊的对象相关联，而每一个成员类与它所在的类的实例却是有关联的。在非顶层成员类内定义一个静态的顶层成员会造成混淆，所以这么做是不被允许的。
- 只有类可以被定义为非静态成员。即使 `static` 关键字被省略掉了，接口、枚举类型以及 `annotation` 类型都还是 `static`。

## 成员类的语法

成员类最重要的特性就是它可以访问它所包在对象中的实例字段与实例方法。在例3-8里，我们可以从 `LinkedList.LinkedIterator()` 构造函数看出这一点：

```
public LinkedIterator() { current = head; }
```

在这个例子里，`head`是`LinkedList`类里的一个字段，我们将它赋值给`LinkedIterator`类的`current`字段。如果我们想要让这些对字段的引用明确地表现出来，该怎么做呢？我们可以试试下面这行程序代码：

```
public LinkedIterator() { this.current = this.head; }
```

此程序将无法被编译。对 `this.current` 的访问并没有错，它明确地引用了新创建的 `LinkedIterator` 对象的 `current` 字段，而 `this.head` 表达式就会发生错误，它实际上引用的是 `LinkedIterator` 对象里一个名为 `head` 的字段。因为在 `LinkedIterator` 对象里根本就没有定义这样的字段，所以编译器当然会产生错误。为了解决这个问题，Java定义了一个特殊的语法，好让你能正确地引用包含实例中的 `this` 对象。因此，如果我们想要让构造函数更明确点，可以使用以下的语法：

```
public LinkedIterator() { this.current = LinkedList.this.head; }
```

该语法的一般形式是 `classname.this`，`classname`是包含类的名称。请注意成员类本身也可以包含有成员类，而且不限嵌套的深度。因为成员类不可以和它的包含类的名称相同，因此该语法中附加在 `this` 之前的类名可用来引用任何的包含实例。此语法只有在它引用包含类中的成员，而该成员的名称又与成员类里的成员的名称相同时才是必要的。

## 访问包含类的超类成员

当一个类隐藏或覆盖了它的超类的成员时，你可以使用 `super` 关键字来引用被隐藏的成员。这个 `super` 语法也一样可以让成员类使用。在少数情况下，当你需要引用包含类 `C` 的超类的隐含字段 `f` 或覆盖超类的方法 `m` 时，可以使用以下的表达式：

```
C.super.f  
C.super.m()
```

## 包含实例的说明

我们都知道，每一个成员类的实例都是与它的包含类的实例相关联的，请再看一下例3-8中所定义的 `iterator()` method：

```
public Iterator<Linkable> iterator() { return new LinkedIterator(); }
```

当成员类的构造函数以这样的方式被调用时，成员类的新实例会自动地与 `this` 对象相关联，这就是你所预期且在大多数的情况之下希望发生的。然而，有时候你会希望在实现成员类时明白地指出其包含实例，这时可以在 `new` 运算符之前加上对包含实例的引用。因此，`iterator()` 方法可以写成如下的样子：

```
public Iterator<Linkable> iterator() { return this.new LinkedIterator(); }
```

现在我们假设我们没有为 `LinkedStack` 定义 `iterator()` 方法。在这样的情况下，`LinkedStack` 对象生成 `LinkedIterator` 对象的程序代码可能会像是这样：

```
LinkedStack stack = new LinkedStack(); // 创建一个空的堆栈
Iterator i = stack.new LinkedIterator(); // 创建一个 Iterator 对象
```

包含实例隐含地说明了包含类的名称，如果你显示地指定包含类的名称，就会发生语法错误：

```
Iterator i = stack.new LinkedStack.LinkedIterator(); // 语法错误
```

另外一个特殊的 Java 语法可以为成员类明确地指定实例。在我们讨论之前，有一点你必须知道：你应该尽量少用到这个语法。这是随着嵌套类型的优异特性而偷偷潜进此语言的一种病态应用。

也许它看起来有点奇怪，但顶层类的确有可能扩展成员类。这表示子类并没有包含实例，但超类却有。当子类的构造函数调用超类的构造函数时，子类的构造函数必须声明它的包含实例，你可以在 `super` 关键字前加上相应包含实例的名称与一个点号来达到这个目的。如果我们没有在 `LinkedIterator` 类中将 `LinkedStack` 的成员声明为 `protected`，就可以扩展它。虽然不清楚为什么要这么做，但我们可以写出如下的程序代码：

```
// 一个顶层类，它扩展了成员类
class SpecialIterator extends LinkedStack.LinkedIterator {
    // 构造函数必须在调用超类的构造函数时，明确地说明它的包含实例
    public SpecialIterator(LinkedStack s) { s.super(); }
    // 省略类的其余部分……
}
```

## 成员类的作用域范围和继承的比较

我们已经知道了顶层类可以扩展一个成员类。在使用非静态成员类时，对任何类都需考虑到两种独立的层次结构。第一种是继承层次结构（inheritance hierarchy），从超类到子类，它定义了成员类所继承的字段与 `method`。第二种是包含层次结构（containment hierarchy），从包含类到被包含类，它定义了成员类作用域范围内（也就是访问权）的字段与 `method`。



这两种层次结构是完全不同的，你最好不要混淆了。你应该避免将超类的字段或method与包含类的字段或method取相同的名称。如果真的发生命名冲突，则所继承的字段或method会将包含类里名称相同的字段或method给覆盖掉。这样的行为是合逻辑的：当类继承了字段或method后，那些被继承的字段或method就会变成类的一部分。因此，被继承的字段与method就在继承它们的类的作用域范围之内了，而且会取代所处作用域范围里相同名称的字段和method。

有一个好方法可以用来防止类层次结构与包含层次结构之间的混淆，就是避免有深的包含层次。如果一个类的嵌套结构超过了两层，则它可能会造成很多的混淆而不值得你这么去做。此外，如果一个类有深的类层次（也就是它有很多的超类），那么你应该考虑将它定义为顶层类。

## 局部类

局部类（local class）是在Java程序块里所局部声明的类。只有类可以被局部定义：接口、枚举类型与annotation类型必须是顶层或静态成员类型。一般来说，局部类是定义在方法里，但它也可以被定义在静态初始化程序或实例初始化程序中。因为所有的Java程序代码块都会出现在类定义里，所以所有的局部类都会被嵌套在包含类中。基于此原因，局部类共享了许多成员类的特性。不过，通常还是会将它视为另一种完全独立的嵌套类型。局部类与成员类间的关系和局部变量与实例变量之间的关系相当接近。

局部类的特性就是它对一个程序块来说是局部的。就和局部变量一样，一个局部类只有在所处程序块的作用域内才是有效的。如果有一个成员类只会被包含类的一个method使用，则没有理由把它写成成员类而不写成局部类。例3-9显示了我们该如何修改LinkedList类的iterator() method来将LinkedListIterator定义为局部类而不是成员类。这么一来，类的定义更接近它被使用的位置，同时也更增强了程序的可读性。为了简化起见，例3-9只显示了iterator() method，而不是整个LinkedList类。

### 例3-9：定义与使用局部类

```
// 此方法返回了LinkedList的Iterator对象
public Iterator<Linkable> Iterator() {
    // 此处将LinkedListIterator定义为一个局部类
    class LinkedListIterator implements Iterator<Linkable> {
        Linkable current;

        // 构造函数使用包含类的private head字段
        public LinkedListIterator() { current = head; }
        // 以下的三个method被Iterator接口所定义
        public boolean hasNext() { return current != null; }
        public Linkable next() {
```

```
        if (current == null) throw new java.util.NoSuchElementException();
        Linkable value = current;
        current = current.getNext();
        return value;
    }
    public void remove() { throw new UnsupportedOperationException(); }
}

// 创建与返回我们刚才定义的类的实例
return new LinkedIterator();
}
```

## 局部类的特性

局部类有以下几个有趣的特性：

- 与成员类一样，局部类与包含实例相关联，而且可以访问任何成员，包括包含类的 `private` 成员。
- 除了访问由包含类所定义的字段之外，局部类还可以访问位于局部方法定义的作用域里的且声明为 `final` 的任何局部变量、`method` 参数或异常参数。

## 局部类的限制

局部类有以下几个限制：

- 局部类的名称只可被定义在定义它的程序块中，它不能在该程序块以外被使用（请注意，局部类的实例可以在类的作用域内被创建，而且在该作用域之外也可以存在。这种情形在稍后的章节将会做更详细的介绍）。
- 局部类不能被声明为 `public`、`protected`、`private` 或 `static`。这些修饰符是针对类成员，它们无法用来声明局部变量或局部类。
- 跟成员类一样，而且基于相同的理由，局部类不可以包含 `static` 字段、`method` 或类。唯一的例外就是常量，它可以同时被声明为 `static` 与 `final`。
- 接口、枚举类型以及 `annotation` 类型无法被局部定义。
- 局部类和成员类一样，它的名称不能跟它的包含类的名称相同。
- 就如我们之前所提到过的，局部类可以使用作用域中的局部变量、成员参数甚至是异常参数，但这些变量或参数必须声明为 `final`。这是因为局部类的实例的生命周期可以比定义它的 `method` 的执行时间长。基于此原因，局部类必须对它所使用的所有局部变量拥有一份私有的内部副本（这些副本是由编译器自动产生的）。要确保局部变量与副本永远一致的方法是将局部变量声明为 `final`。

## 局部类的语法

在 Java 1.0 中, 只有字段、method 与类可以被声明为 final。在 Java 1.1 里, 加入了局部类, 并放宽了 final 修饰符的使用范围, 它可以被用于局部变量、方法参数甚至是一个 catch 语句的异常参数。在这些新用法中, final 修饰符的意义却没有任意的变化: 一旦该局部变量或参数被赋值, 该数值便不能被改变。

局部类的实例就像非静态成员类的实例一样, 会有一个包含类的实例被传给局部类里所有构造函数。局部类可以使用像非静态成员类所用的 this 语法一样, 来明确地引用包含类的成员。因为局部类在该程序块外面是不可见的, 所以不需要像局部类一样, 使用 new 与 super 语法来明确地指出包含实例。

## 局部类的作用域范围

在讨论非静态成员类的时候, 我们知道了成员类可以访问所有继承自超类的成员以及定义于该类内的成员。对局部类来说也是相同的, 但局部类可以访问 final 类型的局部变量与参数。以下的程序代码说明了局部类可以访问的字段与变量:

```
class A { protected char a = 'a'; }
class B { protected char b = 'b'; }

public class C extends A {
    private char c = 'c';           // 对局部类来说, private 字段是可见的
    public static char d = 'd';
    public void createLocalObject(final char e)
    {
        final char f = 'f';
        int i = 0;                  // i 不是 final 类型的, 所以不能被局部类使用
        class Local extends B
        {
            char g = 'g';
            public void printVars()
            {
                // 这些字段与变量都可以被 this 类所访问
                System.out.println(g); // (this.g) g 是 this 类的一个字段
                System.out.println(f); // f 是一个 final 类型的局部变量
                System.out.println(e); // e 是一个 final 类型的局部参数
                System.out.println(d); // (C.this.d) d —— 包含类的字段
                System.out.println(c); // (C.this.c) c —— 包含类的字段
                System.out.println(b); // b 是被 this 类所继承
                System.out.println(a); // a 是被包含类所继承
            }
        }
        Local l = new Local();       // 创建一个局部类的实例,
        l.printVars();               // 并调用它的 printVars() 方法
    }
}
```



## 局部变量的作用域与结束

局部变量定义在一个程序块内，那个程序块就是该变量的作用域。局部域变量在它的作用域之外是不存在的。Java 是一种根据词汇判断作用域 (lexically scoped) 的语言，这表示它的作用域概念是依照源代码编写的方式决定的。任何在大括号内的程序代码，都可以使用定义于该块内的局部变量 (注 10)。

词汇作用域定义了一个可以使用某个变量的源代码程序块，然而，我们常会将作用域想象成具有暂时性——将局部变量想象成在 Java 解释器开始执行程序块时才存在，直到解释器离开该程序块为止。这对局部变量与它的作用域来说是一个合理的思考方式。

然而，对局部类的介绍常常会造成混淆，因为局部类可以使用局部变量，而且局部类的实例的生命周期又比解释器执行该程序块的时间长得多。换句话说，如果你创建了一个局部类的实例，那么该实例在解释器执行完该程序块时不会自动地消失，就像以下的程序代码所显示的：

```
public class Weird {
    // 下面会使用到的静态成员接口
    public static interface IntHolder { public int getValue(); }

    public static void main(String[] args) {
        IntHolder[] holders = new IntHolder[10]; // 拥有 10 个对象的数组
        for(int i = 0; i < 10; i++) {           // 使用循环为数组赋值
            final int fi = i;                   // final 类型的局部变量
            class MyIntHolder implements IntHolder { // 局部类
                public int getValue() { return fi; } // 它使用了 final 类型的变量
            }
            holders[i] = new MyIntHolder();      // 实例化此局部类
        }

        // 因为局部类现在已经超过了作用域范围，所以我们不可以使用它。
        // 但我们已经有了该类的 10 个有效的实例并将它存储于数组中。
        // 局部变量 fi 在这里已经不存在于该作用域内了，但它仍然存在于此 10 个对象中的
        // getValue() method 里。所以调用每一个对象的 getValue() 方法并将其结果输出。
        // 它的输出结果为数字 0 到 9
        for(int i = 0; i < 10; i++) System.out.println(holders[i].getValue());
    }
}
```

这一段程序的行为相当令人惊讶。为了让你有所理解，请记住当解释器进入与离开定义该局部类的程序块时，该局部类的方法的词汇作用域范围并不会做任何的事。还有另一种思考方式：局部类的每一个实例都会为它所使用的 final 类型的局部变量自动地创建一份副本，因此当它被创建时，它会拥有对该作用域私有的一份副本。

注 10： 本章节涵盖了较深的内容，初次接触的读者可以先跳过，等稍微有点概念后再回来继续阅读。

局部类 `MyIntHolder` 有时候被称为是一个结束 (closure)。在一般的术语中, closure 是个对象, 它保存了一个作用域的状态同时让该作用域是可用的。在某些编程风格里, closure 是有用的, 而且不同的编程语言在定义与实现 closure 的方法上是不同的。Java 的 closure 相对较弱些 (而且程序员有时候会为它们不是真的 closure 而争论), 因为它只保留 `final` 变量的状态。

## 匿名类

匿名类 (anonymous class) 是没有名称的局部类, 它是在一个单一的简明表达式中, 使用 `new` 运算符来定义并将它实例化。局部类的定义是 Java 程序块中的一个语句, 而匿名类的定义则是一个表达式, 也就是它可被包含在一个更大的表达式里面, 例如一个方法调用。事实上, 匿名类较局部类更为常见。如果你发现自己定义了一个 `short` 局部类而且只会被使用一次时, 请考虑使用匿名类, 它会将该类的定义与使用放在同一个地方。

请看例 3-10, 它为 `LinkedIterator` 类实现了一个位于 `LinkedStack` 类的 `iterator()` 方法中的匿名类。请将它与例 3-9 进行比较, 可以看出在例 3-9 中同样的类是被作为一个局部类。此范例中的 `generic` 语法会在第四章中做说明。

例 3-10: 使用匿名类来实现 enumeration

```
public Iterator<Linkable> iterator() {  
    // 匿名类被定义为 return 语句的一部分  
    return new Iterator<Linkable>() {  
        Linkable current;  
        // 使用实例初始化函数来取代构造函数  
        { current = head; }  
  
        // 以下三个 method 是被 Iterator 接口所定义  
        public boolean hasNext() { return current != null; }  
        public Linkable next() {  
            if (current == null) throw new java.util.NoSuchElementException();  
            Linkable value = current;  
            current = current.getNext();  
            return value;  
        }  
        public void remove() { throw new UnsupportedOperationException(); }  
    }; // 请注意, 此处需要用分号, 它会结束 return 语句  
}
```

匿名类常见的用法是为适配器类 (adapter class) 提供一个简单的实现。适配器类所定义的程序代码可以被其他的对象所调用, 例如, `java.io.File` 类的 `list()` method。此 method 会将目录内的所有文件列出, 在它返回结果之前, 它会传递每个文件的文件名到 `FilenameFilter` 对象。此 `FilenameFilter` 对象可以接受或拒绝各个文件。当

你实现 `FilenameFilter` 接口时，就是为 `File.list()` method 的使用定义了一个 adapter class。因为这种类的主体一般来说都是非常短的，所以将 adapter class 定义为匿名类是非常简单的事。这里有个例子，让你能定义一个 `FilenameFilter` 类以列出以 .java 结尾的文件：

```
File f = new File("/src");           // 要列出的目录

// 现在以一个 FilenameFilter 自变量来调用 list() method
// 定义与实例化 FilenameFilter 的匿名实现，以此作为该 method 调用表达式的一部分
String[] fileList = f.list(new FilenameFilter() {
    public boolean accept(File f, String s) { return s.endsWith(".java"); }
}); // 不要忘了小括号以及用来结束方法调用的分号！
```

正如你所看到的，定义匿名类与创建该匿名类的实例的语法都使用了 `new` 关键字，然后紧跟着的是类名与由大括号括起来的类主体。如果紧跟在 `new` 关键字后的是类的名称，则该匿名类就是该类的子类。如果在 `new` 之后的是一个接口名称，就像前面的两个例子一样，匿名类就会实现该接口并扩展 `Object`。此语法并不包含任何指定 `extends` 子句、`implements` 子句或是该类的名称的方式。

因为匿名类没有名称，所以在该类内是不能为它定义构造函数的，这是匿名类的基本限制之一。任何在匿名类中所定义的在超类名称之后使用小括号所括起的自变量，都会被传入超类的构造函数中。匿名类一般是用来扩展较简单且没有任何构造函数自变量的类，所以在匿名类定义的语法中，小括号内通常是空的。在之前的范例里，每一个匿名类都实现了一个接口并扩展 `Object`。因为 `Object()` 构造函数没有自变量，所以在范例中的小括号内是空的。

## 匿名类的特性

匿名类可以让你定义一个只会使用一次的类。匿名类拥有局部类的所有特性，而它的语法也相当的简单，因此可以减少程序代码的复杂度。

## 匿名类的限制

因为匿名类只是局部类的一种，所以匿名类与局部类的限制相同。匿名类除了可以定义 `static final` 常量之外，不能定义任何的 `static` 字段、`method` 或类。接口、枚举类型和 `annotation` 类型都不能以匿名的方式来定义。同样地，和局部类一样，匿名类也不能被声明为 `public`、`private`、`protected` 或 `static`。

因为匿名类没有名称，所以无法为匿名类定义构造函数。如果你的类需要构造函数，就必须使用局部类来代替。然而，你可以使用实例初始化函数来替代构造函数。



定义匿名类的语法同时结合了定义与实例化。如果每次在执行该块时便需要创建一个以上的实例，那么此时若用匿名类来代替局部类并不是恰当的做法。

## 匿名类的语法

我们已经看过匿名类在定义与实例化时的语法范例。我们可以更正式地将语法表示为：

```
new class-name ( [ argument-list ] ) { class-body }
```

或：

```
new interface-name () { class-body }
```

虽然在使用匿名类时没有太多的限制，但实例初始化程序却是用来支持匿名类的特殊语法。就像本章稍早讨论过的“字段默认值与初始化程序”，实例初始化程序是在类定义内使用大括号所括起来的初始化程序代码的一个程序块，而实例初始化程序的内容会自动地为该类插入所有的构造函数，包括任何自动创建的默认构造函数在内。因为匿名类并不能定义构造函数，所以它会拥有一个默认的构造函数。通过使用实例初始化程序，你会发现匿名类的确是不能定义构造函数。

## 使用匿名类的时机

正如我们所讨论过的，匿名类的行为和局部类一样，而且它们之间的差异只在于用来定义及产生实例匿名类与局部类的语法不同而已。在程序代码里，当你在匿名类与局部类之间作选择时，这样的决策通常都会成为形式上的问题，也就是你应该选用可以让程序代码更为清楚的语法。一般来说，在以下的情形中你应该考虑使用匿名类：

- 类的主体很短。
- 只需要类的一个实例。
- 类在定义之后马上就会被使用。
- 类的名称并不会让你的程序代码变得更易于了解。

## 匿名类的缩排与格式

大家对于Java和C语言里的一般的缩排与格式应该都非常的熟悉，但一旦要在表达式里放入匿名类的定义，这样的习惯就会被打破。根据 Sun 工程师对嵌套类型的使用经验，他们定义了以下的格式规则：

- 左大括号不能自己单独占一行，它应该紧跟在 new 运算符的右小括号后面。同样地，new 运算符应该尽可能和赋值或其他它所在的表达式出现在同一行。
- 匿名类的主体应该比包含 new 关键字的那一行的开头向内缩排一点。
- 匿名类的右大括号也不应该自己单独占一行，它后面要紧跟着表达式剩余的其他部分，通常是一个分号或是右小括号再接着一个分号。这个额外的分号主要是用来告诉读者这个段落不是一般的程序，同时让匿名类在程序里更容易被理解。

## 嵌套类型的运作方式

前面几节我们介绍了四种嵌套类型的特性与行为，严格来说，我已经向你介绍了需要了解的关于内部类的全部内容。然而，如果你了解它们是如何实现的，那么将有助于对嵌套类型的理解。

嵌套类型是在 Java 1.1 中加入的，不管它对 Java 语言造成了多大的变动，嵌套类型并不会改变 JVM 或 Java 类文件的格式。对 Java 解释器来说，并没有像嵌套类型这样的东西存在着：所有的类都是正常且是顶层类。为了让嵌套类型的行为就像是真的被定义在其他类中一样，Java 编译器会在最后将隐含字段、method 与构造函数自变量插入它所产生的类中。你也许会想要使用 *javap* 反汇编器（disassembler）来分析嵌套类型的类文件，以了解编译器如何让嵌套类型能够运作的技巧（关于 *javap* 的信息请参阅第八章）。

## 静态成员类型的实现

请回想一下我们第一个实现的 *LinkedStack* 范例（例 3-7），它定义了一个名为 *Linkable* 的静态成员接口。当你在编译这个 *LinkedStack* 类时，编译器会产生两个类文件，第一个正如你所预期的是 *LinkedStack.class*，而第二个类文件则是 *LinkedStack\$Linkable.class*，其中 *\$* 是由 Java 编译器自动插入的，而且这两个类文件包含了静态成员接口的实现。

就像我们先前所讨论过的，静态成员类型可以访问它的包含类中的所有 *static* 成员。如果静态成员类型真的这么做了，则编译器就会利用包含类的名称来自动验证成员访问表达式。静态成员类型甚至可以访问它的包含类的 *private static* 字段，但是因为静态成员类型被编译入顶层类，所以它没有办法直接地访问该类的 *private* 成员。因此，如果静态成员类型使用了它的包含类的 *private* 成员，则编译器会产生非 *private* 的访问方法，同时将访问 *private* 成员的表达式转换为调用这些专门产生方法的表达式。这些方法会被给定默认的包访问权，而这样其实就足够了，因为成员类与其包含类都必定会位于同一个包中。

## 非静态成员类的实现

非静态成员类的实现和静态成员类型非常相似，它也会被编译入独立的顶层类文件，而编译器也会执行各种不同的程序代码的修改操作，以让内部类成员的访问操作都能顺利进行。

非静态成员类与静态成员类型最大的不同点是，非静态成员类的实现与其包含类的实现是相关联的，而编译器会为每一个成员类定义一个名为`this$0`的虚拟字段来实施此关联性，而此字段是用来保存对它的包含实例的引用。每一个非静态成员类的构造函数都会包括一个额外的参数，该参数会初始化此字段。每当成员类的构造函数被调用时，编译器会自动地传递一个对它的包含类的引用给此额外的参数。

正如我们所看过的，非静态成员类就像类中的所有成员一样，可以被声明为`public`、`protected`、`private`或指定的默认包类型。就和顶层类一样，成员类被编译成类文件，但顶层类只有`public`或`package`访问权限。因此，对Java解释器来说，成员类只可以为`public`或`package`类型，也就是说声明为`protected`的成员类实际上会被视为是`public`类，而声明为`private`的成员类则会被视为具有与该包相同的类型。但这并不代表你绝对不可以把成员类声明为`protected`或`private`。虽然JVM无法执行这些访问控制修饰符，但这些修饰符会被存储于文件中，Java编译器也会去执行它们。

## 局部类与匿名类的实现

局部类能够引用它的包含类中的字段与方法，而它能如此做的原因就跟成员类一样。它会传递一个隐含的引用给包含类的构造函数，并将该引用存储于由编译器所产生的`private`字段中。同时，与非静态成员类一样，局部类也可以使用它们的包含类中的`private`字段与`method`，因为编译器会插入任何需要的访问`method`。

局部类与成员类的不同之处在于局部类拥有可以引用定义它们的作用域范围内的局部变量。然而，有一个很重要的限制是，局部类只可以引用被声明为`final`的局部变量与参数，而这个限制的原因从实现上来看就会非常清楚。局部类可以使用局部变量是因为编译器会自动地生成`private`实例字段，并拥有该类所会使用到的所有局部变量的副本。编译器也会为每一个局部类的构造函数加入隐含参数，并且这些被创建的`private`字段会自动地为它们做初始值的设定。局部类并无法实际地访问局部变量，它们只是拥有自己的这些变量的副本而已。这样的机制能正确运作的唯一方法是局部变量被声明为`final`，则它们被保证不会更改。有了这样的保证，局部类便可以确保它所拥有的副本永远会与局部变量相同。



因为匿名类没有名称，你也许会对代表它们的类是如何被命名的感到好奇。这是属于实现的细节，但 Sun 的 Java 编译器会使用数字作为匿名类的名称。如果你将例 3-10 的程序代码编译过后，你会发现它会为匿名类产生一个名为 *LinkedStack\$1.class* 的类文件。

## 修饰符一览

我们已经知道，类、接口与它们的成员都可以用一个或一个以上的修饰符关键字来声明，例如 `public`、`static` 与 `final`。表 3-2 列出了 Java 修饰符，并解释了它们所能修饰的 Java 结构，同时也说明了为什么要这么做。请参阅本章前面的“类定义语法”与“字段声明语法”，这和第二章的“method 修饰符”一样。

表 3-2: Java 修饰符

修饰符	用于	意义
abstract	类	类中含有未实现的方法，而且不能被实例化
	接口	所有的接口都为 abstract 类型的，此修饰符在接口声明中是可被省略的
abstract	method	此方法没有主体，主体将由子类来提供。签名后面接着一个分号，包含它的类也必须声明为 abstract
final	类	该类没有任何子类
	method	method 无法被覆盖（同时也不能用动态 method 查找）
	字段	该字段的值无法被更改。定义为 static final 的字段是编译时常量
	变量	局部变量、method 参数或异常参数都无法被更改，对局部类来说相当的有用
native	method	method 是以与某种平台相依的方式来实现的（通常是 C）。没有提供主体，签名后面接着一个分号
无（包）	类	非 public 类只能在它的包中被访问
	接口	非 public 接口只能在它的包中被访问
	method	非 private、protected 或 public 的成员默认为 package 类型，同时可在它的包中被访问
private	method	成员只能在定义它的类中被访问
protected	method	成员只能在定义它的包及子类中被访问
public	类	类可在其他的包中被访问
	接口	接口可在其他的包中被访问
	method	方法可在其他的类中被访问

表 3-2: Java 修饰符 (续)

修饰符	用于	意义
strictfp	类	此类的所有 method 都是 strictfp
strictfp	method	所有由此 method 所运作的浮点数运算必须遵守 IEEE 754 标准。尤其是, 所有的数值, 包括中间的计算结果, 都必须被表示为 IEEE 754 的 float 或 double 类型, 而且不能利用其他额外的精确度或是该平台所能够支持的浮点数格式或硬件。此修饰符很少被用到
static	类	被声明为 static 的内部类是顶层类, 与包含类的成员不相关
	method	static method 是类 method, 它不会获得隐含的 this 对象参数, 它可通过该类的名称被调用
..	字段	static 字段是类字段, 不管有多少类实例被创建, 只能存在一个该字段的实例。它可通过类名来被访问
	初始化函数	初始化函数会在类被加载时执行, 而不是在实例被创建时
synchronized	method	此方法对类或实例执行了相当的修改, 所以必须特别注意以确保两个线程不能同时修改类或实例。对 static 方法而言, 在执行此方法之前, 该类的锁定是必需的。对一个非 static 方法来说, 该对象实例的锁定是必需的
transient	字段	此字段不是该对象的持久状态的一部分, 而它也不能与该对象一起被序列化。关于对象序列化的使用, 请参阅 java.io.ObjectOutputStream
volatile	字段	此字段可以被异步线程所访问, 因此某些优化操作不能在它上面执行。此修饰符有时候可以被用来代替 synchronized, 不过它很少被使用到

## 没有包括在 Java 中的 C++ 特性

本章的脚注指出了 Java 与 C++ 之间的相似点与相异点。Java 与 C++ 之间存在许多相同的概念与特性, 使得 C++ 程序员能够很轻松就学会 Java 语言。然而, 还是有几个 C++ 的特性在 Java 里是没有的。一般来说, Java 是不会采用那些让 C++ 语言变得更为复杂的特性。

C++ 支持同时从多个超类中继承方法的多种实现。虽然这看起来是一个有用的功能, 但它却为语言带来了许多复杂的问题。于是 Java 语言的程序员就使用了接口 (interface)

来避免复杂性。因此，一个 Java 中的类只能继承一个单一超类的 method 实现，但是它却能继承多个接口所声明的 method。

C++ 支持模板 (template) 的使用。举例来说，这让你可以实现一个 Stack 类，然后将 `Stack<int>` 或 `Stack<double>` 实例化以产生两种不同的类型：一个是整数型堆栈而另一个是浮点型堆栈。Java 5.0 引进了参数化类型 (parameterized type) 或泛型 (generic)，它们在一个非常健全的状态下提供了相似的功能。泛型将在第四章做详细的介绍。

C++ 允许你定义运算符来对类实例进行任何运算。实际上，它可以使你扩展语言的语法。这是一个非常棒的特性，我们称它为运算重载 (operator overloading)，它可以写出非常精巧的程序。然而，事实上，它却会使得程序难以理解。经过长期讨论后，Java 语言设计者决定不使用这样的运算重载。但请注意，在 Java 中，将两个字符串连接在一起使用的 + 运算符是一种运算重载。

C++ 允许你对类定义转换函数 (conversion function)，当一个值被赋给此类的变量时，它会自动地调用适当的构造函数方法。这只是一个语法上的便捷方式 (和覆盖赋值运算一样)，而 Java 并不包含它。

在 C++ 里，对象在默认情况使用的都是数值，你必须使用 & 来指定变量或函数自变量自动依引用来操作。在 Java 里，所有的对象都是根据引用来操作，所以不需要 & 语法。







# Java 5.0 新增功能

本章涵盖了Java 5.0版最重要的三个功能。增加了类型安全性 (type-safety) 的泛型 (generics) 以及通过类型参数化使得Java程序更为丰富。例如，一个包含了String对象的List可以被写为List<String>。使用参数化类型可以让Java程序代码更为清楚易懂，同时在程序里可以不需要使用强制转换 (cast)。

枚举类型 (enumerated type)，或称enums，是一个新的引用类型，它和类与接口一样。枚举类型内定义了一组有限的 (枚举) 值，更重要地，它也提供了type-safe：枚举类型的变量只可以拥有该枚举类型内的值或为null。这里有一个定义枚举类型的范例：

```
public enum Seasons { WINTER, SPRING, SUMMER, AUTUMN }
```

本章所讨论到Java 5.0的第三个功能是程序的注释 (annotation) 以及定义它们的注释类型 (annotation type)。注释和程序元素的数据 (或元数据，metadata) 有关，如类、method、字段甚至是method的参数或局部变量。注释内所拥有的数据类型是由它的注释类型所定义的，和枚举类型一样，它是另一种新的引用类型。Java 5.0平台包含了三个标准的注释类型用来提供额外的信息给Java解释器。你将会发现注释被大量用在Java企业级程序设计 (Java enterprise programming) 中的程序代码产生工具里。

Java 5.0也提供了其他一些重要的新功能，它们都不需要特别用一个章节来做解释。这些都在第二章里可以找到，包括

- autoboxing 与 unboxing 的转换
- for/in 循环语句，有时候称为“foreach”
- 拥有不定长度的自变量列表的method，这是我们所知道的 *varargs method*

- 当在覆盖 method 时，可以将该 method 的返回类型给缩小，这是我们所知道的“协变返回 (covariant return)”
- `import static` 指令，它可以将静态成员类型导入为命名空间 (namespace)

## 泛型 (Generic Type)

泛型与 generic method 是 Java 5.0 定义的新功能。泛型被定义为使用了一个或多个类型变量以及使用类型变量的一个或多个 method，就像是自变量或返回类型的占位符 (placeholder)。例如，`java.util.List<E>` 是一个泛型：拥有占位符 `E` 表示的类型的元素的一个列表，此类型有一个名为 `add()` 的 method，它的自变量被声明为类型 `E`；还有一个名为 `get()` 的 method，它的返回值被声明为类型 `E`。

为了要使用泛型，你必须明确地说明类型变量（或变量）的真实类型，于是就会产生像 `List<String>` 这样的参数化类型 (parameterized type) (注 1)。要明确地说明这个特定类型是因为，编译器在编译时会为你提供强大的类型检查能力，同时为你的程序增加 type safety。例如，这样的类型检查可以防止你在只拥有 `String` 对象的列表里加入 `String[]` 数组。编译器会知道 `List<String>` (举例说明) 的 `get()` method 将会返回一个 `String` 对象，而你就不需要将 `Object` 类型的返回值强制转换为 `String`。

在 Java 5.0 中的 `java.util` 包里的 collection 类都被改为泛型，而且在你的程序里将会非常频繁地使用到它。typesafe collection 是泛型的标准使用范例，即使你没有定义你自己的泛型，而且也没有使用 `java.util` 里 collection 类以外的其他泛型，类型安全性 (typesafe) 的 collection 将会带来很大的益处，同时它们也会证明这个新功能如此复杂是非常正常的。

我们从在最基本的 typesafe collection 里使用泛型开始讨论，然后再更仔细地去探讨泛型的使用，并将对类型参数通配符 (type parameter wildcard) 与有限制的通配符 (bounded wildcard) 做深入的说明。在介绍了如何使用泛型之后，我们将会介绍如何编写自己的泛型与 generic method。最后要介绍的是 Java 核心 API 里的一些重要的泛型，除了要探讨这些类型之外——其实这些 API 也已经全面使用了泛型技术，为的就是让我们更深入地了解泛型是如何运作的。

---

注 1： 在本章里，我已经试着将一些专有名词一致化，“generic type”是指一个可以声明为一个或多个类型变量的类型，而“parameterized type”是指一个拥有真实类型参数的可以用来代替类型变量的泛型。然而，一般情况下，它们之间的差异并不十分的明显，而且这些专有名词有时候是可被交换使用的。

## typesafe collection

java.util 包包含了可以与对象列表 (list) 和集合 (set) 一起运作的 Java Collections Framework, 以及从 key 对象到 value 对象的映射 (mapping)。collection 将在第五章做详细的介绍, 在这里, 我们只讨论 Java 5.0 里的 collection 类, collection 类使用了泛型参数来说明 collection 里的对象类型。在 Java 1.4 或更早的版本中, 情况却并非如此。如果没有泛型, 程序员在使用 collection 时就必须记得每一个 collection 的元素类型。当你在 Java 1.4 版本里创建一个 collection 时, 你知道你想要存储在该 collection 里的是哪一种类型的对象, 但编译器并不会知道, 所以你必须非常小心地为它加入适当类型的元素。而且当你在查询 collection 里的元素时, 你必须明确地写出强制转换, 将它们从 Object 转为真实的类型。请看以下的 Java 1.4 版本中的程序代码:

```
public static void main(String[] args) {
    // 此列表只可以用来存储字符串
    // 编译器并不知道, 所以我们必须自己记得
    List wordlist = new ArrayList();

    // 哎呀! 我们加入了 String[] 来代替 String
    // 编译器并不知道这样是错误的
    wordlist.add(args);

    // 因为 List 可以拥有任意的对象, 则 get() method 会返回 Object。
    // 因为列表只能存储字符串, 所以我们将返回值强制转换为 String,
    // 但却产生了 ClassCastException 的异常, 这是因为上一行错误所导致的
    String word = (String)wordlist.get(0);
}
```

此范例说明了泛型可以解决 type safety 的问题。List 与 java.util 里的其他 collection 类都已经被重新定义成泛型了。正如之前所提到过的, List 已经被重新定义成一个名为 E 的类型变量, 且该类型变量为列表元素的类型。add() method 被重新定义为类型 E 的自变量以取代 Object, 而且 get() method 也已经被重新定义为返回 E 来取代 Object。

在 Java 5.0 中, 当我们声明了一个 List 变量或创建 ArrayList 的实例时, 我们指定了所想要的真实类型为 E —— 通过将真实类型置于泛型名称之后的尖括号里。例如, List<String> 就是一个拥有字符串类型的链表。请注意这和将一个自变量传递给 method 非常像, 除了我们使用的是类型而不是值以及使用了尖括号来代替小括号之外。

java.util collection 类的元素必须要是对象, 而这些对象不能使用基本类型值, 接下来要介绍的泛型也无法改变这种情况。泛型无法与基本类型一起运作, 例如, 我们不能这么声明 Set<char> 或 List<int>。然而, 请注意 Java 5.0 的 autoboxing 与 autounboxing 功能使得泛型可以使用 Set<Character> 或 List<Integer>, 就好像可以直接地与 char 和 int 值一起运作一样简单 (请看第二章 autoboxing 与 autounboxing 的详细说明)。



在 Java 5.0 中，上面的范例可以被重新改写成以下的样子：

```
public static void main(String[] args) {
    // 这个列表只可以存储 String 对象
    List<String> wordlist = new ArrayList<String>();

    // args 是一个 String[], 而不是 String, 所以编译器不会让我们这么做
    wordlist.add(args); // 编译错误 !

    // 我们可以这么做
    // 请注意, 使用了新的 for/in 循环语句
    for(String arg : args) wordlist.add(arg);

    // 不需要使用强制转换。List<String>.get() 返回了一个 String
    String word = wordlist.get(0);
}
```

请注意，此程序代码并不比非泛型的范例来得短。这里使用了类型参数将 String 放在尖括号里，以取代于小括号内放入 String 的强制转换。不同之处是该类型参数只被声明一次而已，但该列表可以被使用很多次而不需要做强制转换。这在较长的范例里将会更明显。即使泛型语法比非泛型语法更为冗长，但使用泛型还是非常值得的，因为额外的类型信息允许编译器在你的程序代码上执行强大的错误检查。错误本来只会在运行时才会出现，而现在在编译期就可以被检测出来了。此外，编译错误也会精确地指出哪一行违反了 `type safety`。若不使用泛型，则产生错误的程序来源就会抛出 `ClassCastException`。

就和 `method` 可以有很多自变量一样，类也可以有超过一个以上的类型变量，`java.util.Map` 接口就是一个例子。`Map` 是 `key` 对象与 `value` 对象之间的映射 (mapping)。`Map` 接口声明了一个类型变量用来表示 `key` 的类型以及一个变量用来表示 `value` 的类型。就如以下的范例，假设你想要将 String 对象映射到 Integer 对象：

```
public static void main(String[] args) {
    // 一个从字符串到它们在 args[] 数组里的位置的映射
    Map<String,Integer> map = new HashMap<String,Integer>();

    // 请注意, 我们使用 autoboxing 来将 i 封装成 Integer 对象
    for(int i=0; i < args.length; i++) map.put(args[i], i);

    // 查找由单词组成的数组索引。请注意, 这不需要使用到强制转换!
    Integer position = map.get("hello");

    // 我们也需要依赖 autounboxing 将它直接转换为 int,
    // 但是如果该值不存在于 map 中时, 就会抛出 NullPointerException
    int pos = map.get("world");
}
```

一个像 `List<String>` 这样的参数化类型本身就是一个类型了，同时它可以像类型参数的值一样被其他的类型所使用。你会看到像下面这样的程序代码：

```
// 请看这些嵌套的尖括号!
Map<String, List<List<int[]>>> map = getWeirdMap();

// 编译器会知道所有的类型, 同时我们可以写出如下不需要强制转换的表达式。
// 当然, 在运行时仍然会发生 NullPointerException 或 ArrayIndexOutOfBoundsException 异常
int value = map.get(key).get(0).get(0)[0];

// 以下是会发生错误的表达式
List<List<int[]>> listOfLists = map.get(key);
List<int[]> listOfIntArrays = listOfLists.get(0);
int[] array = listOfIntArrays.get(0);
int element = array[0];
```

在上面的程序代码里, `java.util.List<E>` 的 `get()` method 与 `java.util.Map<E,V>` 会返回一个列表或类型 `E` 与 `V` 各自的映射的元素。然而, 泛型可以用更复杂的方式来使用它们的变量。`List<E>` 的 `iterator()` method 的返回值被声明为 `Iterator<E>`, 也就是该 method 会返回一个参数化类型的实例, 且它的实际类型参数会与列表的实际类型参数一样。为了要更具体地说明, 这里有一个方式可以取得 `List<String>` 的第一个元素, 而不是使用 `get(0)` 来调用它。

```
List<String> words = // .....已经在别处初始化.....
Iterator<String> iterator = words.iterator();
String firstword = iterator.next();
```

## 理解泛型

这小节将更深入地探讨泛型使用的细节, 同时会说明以下主题:

- 使用不具有类型参数的泛型的结果
- 参数化类型的层次
- 泛型的类型安全性在编译时的漏洞以及用来确定运行时类型安全性的修补程序
- 为什么参数化类型组成的数组不具有类型安全性

## 原始类型与未校验的警告

虽然 Java 的 collection 类已经被修改成使用泛型的方式, 但你并不需要明确地指出类型参数来使用它们。没有使用类型参数的泛型称为原始类型 (raw type)。现有的 Java 5.0 版以前的程序代码还可以继续运行: 如果在已经写好的程序代码里加上强制转换, 就会给编译器带来很大的麻烦。考虑以下将混合类型的对象存储于原始 `List` 中的程序代码:

```
List l = new ArrayList();
l.add("hello");
l.add(new Integer(123));
Object o = l.get(0);
```

上述的程序代码在 Java 1.4 里可以正常运行。如果我们使用 Java 5.0 的 *javac* 来编译这些程序代码的话，却会显示出这样的错误信息：

```
Note: Test.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

当我们加上 *-Xlint* 重新编译的话，将会看到这些警告信息：

```
Test.java:6: warning: [unchecked]
  unchecked call to add(E) as a member of the raw type java.util.List
    l.add("hello");
    ^
Test.java:7: warning: [unchecked]
  unchecked call to add(E) as a member of the raw type java.util.List
    l.add(new Integer(123));
    ^
```

编译器会在调用 *add()* 处警告我们，它不能保证加入到列表里的值是正确的类型。这是要让我们知道，因为我们使用的是原始类型，所以它无法保证程序代码是具有类型安全性的。请注意，这在调用 *get()* 时是对的，因为它所取得的元素是已经安全地存在于列表中的元素。

如果你在没有使用任何 Java 5.0 新功能的文件里得到了未校验的警告，那么你可以加上 *-source 1.4* 标记来编译那些文件，如果你并不想要加上这个标记的话，那么你就可以忽略这些警告，而使用 *@SuppressWarnings("unchecked")* 注释（请看本章稍后的“注释”一节）来隐藏这些警告信息或修改你的程序代码，以明确地指定该类型参数（注2）。以下的范例在编译后不会再出现警告信息，而且还允许你在该列表中加入混合类型的对象：

```
List<Object> l = new ArrayList<Object>();
l.add("hello");
l.add(123);                // autoboxing
Object o = l.get(0);
```

## 参数化类型的层次结构

参数化类型构成了类型的层次结构，就像一般的类型一样。然而，该层次结构是建立在基本类型上，而不是在参数类型上。你可以试着用以下的例子来做实验：

```
ArrayList<Integer> l = new ArrayList<Integer>();
List<Integer> m = l;                                     // OK
Collection<Integer> n = l;                             // OK
```

注2： 在编写这段时，*javac* 尚未支持 *@SuppressWarnings* annotation，预计在 Java 5.1 中才会支持。



```
ArrayList<Number> o = l; // 错误
Collection<Object> p = (Collection<Object>)l; // 错误, 即使使用了强制转换
```

List<Integer>是一个Collection<Integer>, 而不是List<Object>。了解为什么泛型是以这种方式运作是非常重要的, 而不是仅凭直觉就可以的。请看以下的程序代码:

```
List<Integer> li = new ArrayList<Integer>();
li.add(123);

// 此行程序代码无法编译,
// 但是为了实验, 我们就假设它是可以编译的, 而且会从中发现很多的问题
List<Object> lo = li;

// 现在我们可以将列表的元素恢复为Object以取代Integer
Object number = lo.get(0);

// 但这行可以正常运行吗?
lo.add("hello world");

// 如果上面这行程序代码可以正常运行, 底下这行程序代码就会抛出ClassCastException异常
Integer i = li.get(1); // 无法将String强制转换为Integer!
```

这就是List<Integer>不可为List<Object>的原因, 即使List<Integer>的所有元素实际上都是Object的实例。如果可以将List<Integer>转为List<Object>, 那么非Integer对象就可以被加入列表中。

### 运行时的类型安全性

就像我们所知道的, 即使在X可以被转换为Y的情况下, List<X>也无法被转换为List<Y>。然而, List<X>可以被转换为List, 所以你可以将它传递给原有的method, 该method所期望的是此类型的自变量而且它也没有被修改为泛型。

将参数化类型转换为非参数化类型的能力对向下兼容来说是必要的, 但这的确在泛型所提供的类型安全性系统中开了一个口:

```
// 这是个基本的参数化列表
List<Integer> li = new ArrayList<Integer>();

// 将参数化类型指定给非参数化变量 (nonparameterized variable) 是合法的
List l = li;

// 此行程序代码有个错误, 但它还是可以编译及运行
// Java 5.0 编译器将会产生一个未校验的警告信息
// 如果它出现了与使用Java 1.4 编译后所产生的类一样的结果,
// 那么我们将不会得到警告的信息
l.add("hello");

// 此行在编译后并没有警告信息产生, 但在运行时会抛出ClassCastException异常
// 请注意, 将不会有错误产生
Integer i = li.get(0);
```

泛型只提供编译时的类型安全性，如果你使用了Java 5.0编译器来编译你所有的程序代码，就不会产生任何未校验的警告信息，这些编译时的检查就已经足够保证你的程序代码在运行时也是具有类型安全性。但如果你的程序代码有未校验的警告信息或使用原有的程序代码把collection当成原始类型（raw type）来操作时，你可能要使用额外的步骤来确保运行时的类型安全性。你可以使用像java.util.Collections里的checkedList()与checkedMap() method。这些method会将你的collection封装为wrapper collection，且会利用运行时的类型检查来确保只有正确类型的值才可以被加入collection中。例如，以下的程序代码可以防止类型安全性上的漏洞：

```
// 这是一个基本的参数化列表
List<Integer> li = new ArrayList<Integer>();

// 在运行时类型安全性会被隐藏起来
List<Integer> cli = Collections.checkedList(li, Integer.class);

// 现在将已校验列表扩展至原始类型
List l = cli;

// 此行可以编译，但在运行时会产生ClassCastException的错误
// 异常会正好出现在程序错误处，不会距离太远
l.add("hello");
```

## 参数化类型的数组

在与泛型一起运作时，数组需要有特别的考虑。还记得如果T是S的超类（或接口），那么由类型S[]组成的数组也会是由类型T[]组成的。就是因为这样，Java解释器每次将对象存储于数组时，都必须完成运行时的检查，以确保对象与数组在运行时的类型是兼容的。例如，以下的程序代码在运行时的检查会失败，而且会抛出ArrayStoreException异常：

```
String[] words = new String[10];
Object[] objs = words;
objs[0] = 1; // 1被自动封装成Integer并抛出ArrayStoreException异常
```

虽然在编译时objs的类型是Object[]，但它在运行时的类型却是String[]，所以若将它们存储为Integer是不合法的。

当我们使用泛型时，对数组存储异常的运行时检查就不再足够了，因为在运行时执行的检查无法访问编译时的类型参数信息。考虑这段（假想的）程序代码：

```
List<String>[] wordlists = new ArrayList<String>[10];
ArrayList<Integer> ali = new ArrayList<Integer>();
ali.add(123);
Object[] objs = wordlists;
objs[0] = ali; // 不会有ArrayStoreException异常
String s = wordlists[0].get(0); // ClassCastException!
```

如果以上的程序代码是被允许的话,运行时数组存储的检查就会成功:在没有编译时类型参数的情况下,程序代码可以将ArrayList存储于ArrayList[]数组内,这是完全合法的。由于编译器无法防止你用这种方式破坏类型安全性,所以它改为避免你创建由任何参数化类型所组成的数组。以上的场景不可能会发生,因为编译器会拒绝编译第一行。

请注意,数组使用泛型时并不是没有任何限制的,在创建参数化类型的数组时就有一些限制。当我们讨论如何去编写 generic method 时,就会再回到这个议题。

## 类型参数通配符

假设我们想要写一个可以显示List元素的method(注3)。在List是泛型之前,我们只要把程序代码写成以下形式:

```
public static void printList(PrintWriter out, List list) {
    for(int i=0, n=list.size(); i < n; i++) {
        if (i > 0) out.print(", ");
        out.print(list.get(i).toString());
    }
}
```

在Java 5.0中List是泛型,如果我们试着要去编译这个method,将会产生未校验的警告信息。为了要消除这些警告信息,你可能会想要将method修改成如下形式:

```
public static void printList(PrintWriter out, List<Object> list) {
    for(int i=0, n=list.size(); i < n; i++) {
        if (i > 0) out.print(", ");
        out.print(list.get(i).toString());
    }
}
```

以上的这些程序代码不会出现警告,但并没有太大的用处,因为唯一可以传递给它的列表是被明确地声明为typeList<Object>的列表。一定要记得List<String>与List<Integer>(举例)不能被放大或强制转换为List<Object>。我们真正想要的是一个具有类型安全性的printList() method,可以传递任何List给它,而不必去管它是如何被参数化的。解决方法是使用通配符来表示类型参数。该method可以被写成:

---

注3: 此章节里的三个printList() method忽略了java.util里的List实现类提供了可用的toString() method的事实。也请注意,这些method假设List实现了RandomAccess,同时在LinkedList实例上表现了非常差的性能。



```
public static void printList(PrintWriter out, List<?> list) {
    for(int i=0, n=list.size(); i < n; i++) {
        if (i > 0) out.print(", ");
        Object o = list.get(i);
        out.print(o.toString());
    }
}
```

上述的 method 在编译过后不会出现警告信息，而且可以依据我们想要使用的方式来使用通配符。通配符`?`表示一个未知的类型，`List<?>`类型可以读成“未知类型的`List`”。

就像一般的规则一样，如果有个类型是泛型，而你不知道或你并不在乎该类型变量的值时，你应该使用通配符来代替原始类型。原始类型只有在向下兼容时是被允许的，而且也只可以在之前版本的程序代码中使用。请注意，在调用构造函数时，是不可以使用通配符的。以下就是不合法的程序代码：

```
List<?> l = new ArrayList<?>();
```

创建未知类型的 `List` 是不合理的。如果你创建了一个未知类型的 `List`，就应该知道它将要存储什么样的元素。之后你可能会想要传递这种类型的列表给并不在意它的元素类型的 method，但是当你在创建它时，就需要明确地说明该元素的类型。如果你想要的是可以拥有任何对象类型的 `List` 时，可以这么写：

```
List<Object> l = new ArrayList<Object>();
```

从以上的 `printList()` 变体应该能清楚看出，`List<?>` 与 `List<Object>` 是不同的，而它和原始列表（raw `List`）也是不同的。`List<?>` 有两个非常重要的特性（property），该特性是由于使用通配符而产生的。第一，请看像 `get()` 这样的 method，它被声明为返回一个和类型参数相同类型的值。在这样的情况下，它是一个未知的类型，所以这些 method 会返回 `Object` 类型。因为我们要做的只是调用它的 `toString()` method，所以这还符合我们的需求。

第二，考虑 `List` 中像 `add()` 这样的 method，它被声明为接受一个由类型参数所指定的类型的自变量。这是一个较让人惊讶的状况：当类型参数是未知时，编译器不会让你调用任何有未知类型参数的 method，因为它并不会检查你正在传递的值。`List<?>` 会被有效率地只读，因为编译器不允许我们调用像 `add()`、`set()` 以及 `addAll()` 这样的 method。

## 有限制的通配符

我们现在来看一下原先的范例，但会有稍微复杂的变形。假设我们想要写一个 `sumList()` method 来计算列表里 `Number` 对象的总和。就像之前所看到的，我们可以使用原始列表，但必须舍弃类型安全性，而且必须应付编译器所产生的未校验的警告信

息。或者，我们可以使用 `List<Number>`，但 `List<Integer>` 或 `List<Double>` 不能够调用此 `method`，而事实上类型是非常有可能使用到的。但是如果使用了通配符，将无法真正地获得我们所想要的类型安全性，因为我们必须猜测被 `List` 调用的方法，它的类型参数是 `Number` 或是它的一个子类，而不是 `String`。这里有一个类似的 `method`：

```
public static double sumList(List<?> list) {
    double total = 0.0;
    for(Object o : list) {
        Number n = (Number) o; // 需要强制转换，而且可能会失败
        total += n.doubleValue();
    }
    return total;
}
```

为了要修正以上的 `method` 并让它真正具有类型安全性，我们必须使用有限制的通配符。该 `List` 的类型参数是一个未知的类型，它不是 `Number` 类型就是 `Number` 的子类。以下的程序代码就是我们所想要的：

```
public static double sumList(List<? extends Number> list) {
    double total = 0.0;
    for(Number n : list) total += n.doubleValue();
    return total;
}
```

`List<? Extends Number>` 类型可以被读成“未知类型的 `List` 类是继承自 `Number` 类的”。这非常重要，你必须要了解在相关环境中，`Number` 会被认为是它自己的后代。

请注意，这里已不再需要强制转换。我们不知道列表元素的类型，但知道它们有 `Number` 的“上限”，所以我们可以把它们当成 `Number` 对象而从列表中取出。使用 `for/in` 循环会将取得列表元素的过程隐藏起来。一般规则就是当你在使用通配符且为它设定上限时，`method`（像是 `List` 的 `get()` `method`）的返回值类型将会与上限一样。所以，如果我们是调用 `list.get()` 来取代 `for/in` 循环，我们还是可以获得 `Number` 类型的值。对于调用像 `list.add()` 这样具有类型参数作为自变量的 `method` 的禁止仍然成立。例如，如果编译器允许我们去调用这些 `method`，我们就可以在声明只可以拥有 `Short` 值的列表里加入 `Integer` 值。

我们也可以指定通配符的下限，也就是使用 `super` 关键字来取代 `extends`。这样的技术在能被调用到的 `method` 上会产生不同的影响。下限型通配符的使用比上限型通配符来得少，我们会在本章的稍后讨论。

## 编写泛型与 `method`

创建一个简单的泛型是非常容易的。首先，在类或接口的名称后面声明你的类型变量，

且该类型变量必须放在尖括号内并用逗号来将它们分隔开。你可以在任何的地方使用这些类型变量，任何的实例字段或类 method 中都会需要使用类型。虽然这样，但请记住，该类型变量只有在编译时才会存在，所以类型变量不能与运行时运算符 instanceof 和 new 一起使用。

我们用一个简单的泛型范例作为本节的开场白，而后将会让它变得更完美。这个程序代码定义了一个 Tree 数据结构，它使用了类型变量 V 来代表树的每个节点所拥有的值的类型：

```
import java.util.*;

/**
 * 一个拥有类型 V 的值的树状数据结构
 * 每一棵树都有一个类型 V 的单一值，而且可以有任意数目的分支，
 * 且每一个分支都还是 Tree 的一部分
 */
public class Tree<V> {
    // 此树的值是属于类型 V
    V value;

    // Tree<V> 可以有分支，每个分支也是 Tree<V>
    List<Tree<V>> branches = new ArrayList<Tree<V>>();

    // 这是构造函数。请注意类型变量 V 的用法
    public Tree(V value) { this.value = value; }

    // 以下是操作这些节点值与分支的实例 method
    // 请注意类型变量 V 在自变量或返回类型中的用法
    V getValue() { return value; }
    void setValue(V value) { this.value = value; }
    int getNumBranches() { return branches.size(); }
    Tree<V> getBranch(int n) { return branches.get(n); }
    void addBranch(Tree<V> branch) { branches.add(branch); }
}
```

你或许注意到，类型变量的命名惯例是使用单一的大写字母。使用单一的字母是为了要和实际类型的名称有所区别，因为实际类型一定会有较长、较具描述性的名称。使用大写字母与类型的命名惯例一致，主要是与使用小写的局部变量、method 参数以及字段的名称进行区分。像 java.util 里的 Collection 类通常会使用类型变量 E 作为“元素类型”。当类型变量可以完全地表示任何东西时，T（代表类型）与 S 最常被用来作为泛型变量名称（就像使用 i 与 j 作为循环变量）。请注意，声明为泛型的类型变量只可以被实例字段以及该类型的 method（以及套嵌类型）所使用，而静态字段与 method 是不可以使用泛型的。当然，这是因为它是已被参数化的泛型的实例。静态成员可被所有的实例与类的参数化共享，所以静态成员不会有与它们相关联的类型参数。但是，method（包括静态 method）可以声明并使用它们自己的类型参数，而且这些 method 的调用都可以被参数化。本章的稍后将会说明。



## 类型变量的界限

上述的 `Tree<V>` 类里所声明的类型变量 `V` 是未受限制的, `Tree` 可以使用任何类型来被参数化。我们常想要在可被使用的类型上做些限制: 我们可能会想强制让类型参数实现一个或多个接口, 或者指定该类型是特定类的子类。这可以通过指定类型变量的界限 (bound) 来完成。我们已经看过了通配符的上限, 使用类似的语法也可以指定类型变量的上限。以下的程序代码是一个 `Tree` 的范例, 它已被改写以让 `Tree` 对象具有 `Serializable` 与 `Comparable`。为了这么做, 该范例使用了类型变量界限来确保它的值的类型也是 `Serializable` 与 `Comparable`。请注意在 `V` 上增加的 `Comparable` 界限是如何让我们通过保证 `compareTo()` method 在 `V` 上的存在物来编写 `Tree` 的 `compareTo()` method (注4)。

```
import java.io.Serializable;
import java.util.*;

public class Tree<V extends Serializable & Comparable<V>>
    implements Serializable, Comparable<Tree<V>>
{
    V value;
    List<Tree<V>> branches = new ArrayList<Tree<V>>();

    public Tree(V value) { this.value = value; }

    // 实例method
    V getValue() { return value; }
    void setValue(V value) { this.value = value; }
    int getNumBranches() { return branches.size(); }
    Tree<V> getBranch(int n) { return branches.get(n); }
    void addBranch(Tree<V> branch) { branches.add(branch); }

    // 此method是Comparable<Tree<V>>的非递归实现
    // 它只会比较这个节点的值并会忽略掉分支
    public int compareTo(Tree<V> that) {
        if (this.value == null && that.value == null) return 0;
        if (this.value == null) return -1;
        if (that.value == null) return 1;
        return this.value.compareTo(that.value);
    }

    // 如果我们忽略了在Serializable类里的这个字段, javac -Xlint 就会警告我们
    private static final long serialVersionUID = 833546143621133467L;
}
```

类型变量的界限是在类型变量的名称之后加上 `extends` 以及类型 (它们自己可能会被

注4: 在这里所显示的界限需要类型 `V` 能与它自己做比较, 换句话说, 它直接实现了 `Comparable` 接口。这排除了使用从超类继承的 `Comparable` 接口的类型。我们会在本节的结尾更仔细地查看 `Comparable` 接口。

参数化, 就像 Comparable 一样) 的列表。请注意, 像这样有多个界限的情况, 界限类型会被 & 符号分隔开, 而不是逗号。逗号是用来分隔类型变量, 如果也用逗号来分隔类型变量界限, 就会造成混淆。一个类型变量可以有任意数量的界限, 其中包括任意数量的接口以及最多一个类。

## 泛型中的通配符

我们在本章的前面已经看过在操作参数化类型的 method 里使用通配符以及有限制的通配符的范例。它们在泛型里是非常有用的, 我们目前设计的 Tree 类就需要每一个节点都有相同类型 V 的值。或许这太严格了, 我们应该允许树的分支可以有是 V 的子类型的值, 而不必一定是 V 本身的。此版本的 Tree 类 (没有 Comparable 与 Serializable 的实现) 较具灵活性:

```
public class Tree<V> {
    // 这些字段保存值与分支
    V value;
    List<Tree<? extends V>> branches = new ArrayList<Tree<? extends V>>();

    // 这是个构造函数
    public Tree(V value) { this.value = value; }

    // 这些是操作值与分支的实例 method
    V getValue() { return value; }
    void setValue(V value) { this.value = value; }
    int getNumBranches() { return branches.size(); }
    Tree<? extends V> getBranch(int n) { return branches.get(n); }
    void addBranch(Tree<? extends V> branch) { branches.add(branch); }
}
```

例如, 对于分支类型使用有限制的通配符能让我们加入 Tree<Integer>, 就像是 Tree<Number> 的一个分支一样:

```
Tree<Number> t = new Tree<Number>(0); // 请注意autoboxing
t.addBranch(new Tree<Integer>(1));    // int 1被封装为 Integer
```

如果我们使用 getBranch() method 来查询分支, 则被返回分支的值的类型是未知的, 我们必须使用通配符来表示。以下的两行程序代码是合法的, 但第三行就不是:

```
Tree<? extends Number> b = t.getBranch(0);
Tree<?> b2 = t.getBranch(0);
Tree<Number> b3 = t.getBranch(0); // 编译错误
```

当我们以这样的方式来查询分支时, 我们不知道该值的正确类型, 但我们的确有值的类型的上限条件, 所以我们可以这样做:

```
Tree<? extends Number> b = t.getBranch(0);
Number value = b.getValue();
```

然而，我们却无法设定分支的值或为该分支加上一个新的分支。就像本章的前面介绍过的，虽然有上限的存在，但还是无法改变值的类型未知的事实。编译器并没有足够的信息来让我们安全地将值传递给 `setValue()` 或将新的分支（包含值的类型）传递给 `addBranch()`。以下两行程序代码是不合法的：

```
b.setValue(3.0); // 不合法，值的类型是未知的
b.addBranch(new Tree<Double>(Math.PI));
```

此范例说明了在泛型设计里一般的交换条件：使用有限制的通配符可以让数据结构更有灵活性，但却会降低安全地使用其中一些 `method` 的能力。不管这是不是一个好的设计，它都取决于相关环境。一般来说，泛型要设计得很好是很困难的。但幸运的是在大部分的情况下我们常去使用事先存在于 `java.util` 包里的泛型，而较少使用自己创建的。

### generic method

就如先前所提到的，泛型的类型变量只可以用于该类型的实例成员，而不可以用于静态成员。然而，跟实例 `method` 一样，静态 `method` 也可以使用通配符。虽然静态 `method` 无法使用它们所处类的类型变量，但它们可以声明自己的类型变量。当一个 `method` 声明它自己的类型变量时，它就叫做 `generic method`。

这里有一个可以被加入 `Tree` 类中的静态 `method`。它虽然不是一个 `generic method`，但使用了与我们在本章的前面所看到的 `sumList()` `method` 非常像的有限制的通配符：

```
/** 以递归方式计算树上所有节点的值的总和 */
public static double sum(Tree<? extends Number> t) {
    double total = t.value.doubleValue();
    for(Tree<? extends Number> b : t.branches) total += sum(b);
    return total;
}
```

通过声明类型变量来将此 `method` 改写成 `generic method`，以表示通配符的上限：

```
public static <N extends Number> double sum(Tree<N> t) {
    N value = t.value;
    double total = value.doubleValue();
    for(Tree<? extends N> b : t.branches) total += sum(b);
    return total;
}
```

泛型版本的 `sum()` 并不会比通配符版本的 `sum()` 来得简单，而且类型变量的声明并不会让我们获得任何东西。在这个范例中，典型的通配符的解决方案更优于泛型的解决方案。当类型变量是用来表示两个参数或一个参数与一个返回值之间的关系时，就需要使用到 `generic method`。以下的 `method` 是个范例：



```
// 此 method 会返回两棵树中最大的那一棵树，
// 树的大小是用 sum() method 来计算的。
// 类型变量会确保这两棵树有相同的值的类型，而且都可以被传递给 sum()
public static <N extends Number> Tree<N> max(Tree<N> t, Tree<N> u) {
    double ts = sum(t);
    double us = sum(u);
    if (ts > us) return t;
    else return u;
}
```

此 method 使用了类型变量 `N` 来表示限制条件——自变量与返回值具有相同的类型参数，而且那个类型参数为 `Number` 或它的子类。

或许有人会认为，把两个自变量限制为具有相同的值的类型太过严苛，而且我们应该可以调用 `Tree<Integer>` 与 `Tree<Double>` 的 `max()` method。有一个方式可以表示这个，就是使用两个不相关的类型变量来代表两个不相关的值的类型。然而请注意，我们不能使用该 method 所返回的任何一种类型变量，而必须使用通配符：

```
public static <N extends Number, M extends Number>
    Tree<? extends Number> max(Tree<N> t, Tree<M> u) {...}
```

因为这两个类型变量 `N` 与 `M` 是完全不相关的，而且由于每一个都被置于签名中的单独地方，它们并不优于有限制的通配符。此 method 最好写成这样：

```
public static Tree<? extends Number> max(Tree<? extends Number> t,
                                           Tree<? extends Number> u) {...}
```

所有显示于此的 generic method 范例都是静态 method。这并不是必要的条件，实例 method 也可以声明它们自己的类型变量。

## 调用 generic method

在使用泛型时，你必须指定实际类型参数来取代其类型变量。对 generic method 来说，同样的情况未必成立：编译器几乎总是可以依据你传递给该 method 的自变量来了解 generic method 的正确参数化。例如，考虑上面定义的 `max()` method：

```
public static <N extends Number> Tree<N> max(Tree<N> t, Tree<N> u) {...}
```

在调用这个 method 时不需要指定 `N`，因为 `N` 会被内含地在 method 自变量 `t` 与 `u` 的值中指定。例如，在以下的程序代码中，编译器判定 `N` 是 `Integer`：

```
Tree<Integer> x = new Tree<Integer>(1);
Tree<Integer> y = new Tree<Integer>(2);
Tree<Integer> z = Tree.max(x, y);
```

对 generic method 来说, 编译器用来判定类型参数的过程就叫类型推断 (type inference)。类型推断靠直觉就可以了解, 但编译器真正使用的算法就非常复杂了, 这超出了本书的范围, 在《Java Language Specification》第三版第十五章有完整的细节介绍。

我们来看一下较复杂版本的类型推断, 考虑以下的 method:

```
public class Util {  
    /** 将所有a的元素设定为值v, 返回a */  
    public static <T> T[] fill(T[] a, T v) {  
        for(int i = 0; i < a.length; i++) a[i] = v;  
        return a;  
    }  
}
```

这里有两个 method 的调用:

```
Boolean[] booleans = Util.fill(new Boolean[100], Boolean.TRUE);  
Object o = Util.fill(new Number[5], new Integer(42));
```

在第一个调用里, 编译器可以轻易地判定 T 是 Boolean; 第二个调用里, 编译器判定 T 为 Number。

在极少数的情况下, 你需要为 generic method 明确地指定类型参数。有时这是必要的, 例如, 当 generic method 不需要自变量时。请看 `java.util.Collections.emptySet()` method, 它会返回不具任何元素的集合 (set), 但和 `Collections.singleton()` method 不一样, 它不会接收会指定被返回集合的类型参数的自变量。你可以在 method 名称前面将类型参数放入尖括号内, 以明确地指定类型参数:

```
Set<String> empty = Collections.<String>emptySet();
```

类型参数不能与未限定的 method 名称一起使用, 它们必须在点号或 new 关键字之后, 或是在构造函数中的 this 或 super 关键字之前。

结果就是如果你将 `Collections.emptySet()` 的返回值赋值给一个变量, 就如我们在前面所做的, 类型推断的机制就可以用变量的类型来推断类型参数。虽然在以上程序代码中明确的类型参数规范能作为有用的说明, 但那是不必要的, 而程序代码可以改写为:

```
Set<String> empty = Collections.emptySet();
```

当你在一个 method 调用表达式里使用 `emptySet()` method 的返回值时, 明确的类型参数就是必要的。例如, 假设你想要调用名称为 `printWords()` 的 method, 且该 method 预期会有属于 `Set<String>` 类型的单独自变量。如果你想要传递一个空的 set 给这个 method, 可以用这段程序代码:

```
printWords(Collections.<String>emptySet());
```

在这种情况下，类型参数的明确规范 `String` 就是必要的。

## generic method 与数组

我们在本章的前面已看到编译器不允许你创建一个类型被参数化的数组。然而，并不是把数组和泛型一起使用就会有这个限制。例如，考虑之前所定义的 `Util.fill()` method。它的第一个自变量及返回值都属于类型 `T[]`。此 method 的主体不必创建元素类型为 `T` 的数组，所以此 method 是完全合法的。

如果你写了一个可以使用 `varargs`（请看第二章的“不定长度自变量列表”一节）与类型变量的 method 时，要记得调用 `varargs method` 来执行数组的创建。考虑以下的 method：

```
/** 返回最大的指定值，如果没有则返回 null */  
public static <T extends Comparable<T>> T max(T... values) { ... }
```

你可以调用带有 `Integer` 类型的参数的 method，因为编译器会在你调用该 method 时帮你插入必要的创建数组的程序代码。但如果你要将相同的自变量强制转换为类型 `Comparable<Integer>` 时，你就不能调用此 method，因为创建一个类型为 `Comparable<Integer>[]` 的数组是不合法的。

## 参数化异常

异常是在运行时被抛出与捕获的，而且对编译器来说没有方法可以执行类型检查以确保该不明原因的异常与 `catch` 子句里所指定的类型参数是相匹配的。就是因为这个原因，`catch` 子句不可以包含类型变量或通配符。因为在运行时所捕获的异常与编译时的类型参数不可能完全一样，所以你不能编写出任何继承自 `Throwable` 泛型的子类。类型参数化的异常是不被允许的。

然而，你可以在 method 签名的 `throws` 子句里使用类型变量。例如，考虑下面这段程序代码：

```
public interface Command<X extends Exception> {  
    public void doit(String arg) throws X;  
}
```

此接口用来表示一个“command”：拥有一个单一字符串自变量的程序块而且没有返回值。此程序代码会抛出以类型参数 `X` 表示的异常。这里有一个范例，它使用该接口的类型参数化：

```
Command<IOException> save = new Command<IOException>() {  
    public void doit(String filename) throws IOException {
```



```
        PrintWriter out = new PrintWriter(new FileWriter(filename));
        out.println("hello world");
        out.close();
    }
};

try { save.doit("/tmp/foo"); }
catch(IOException e) { System.out.println(e); }
```

## 泛型案例研究：Comparable 与 Enum

Java 5.0 里新的泛型功能被使用在 Java 5.0 API 中，最主要的是 `java.util`、`java.lang`、`java.lang.reflect` 与 `java.util.concurrent`。这些 API 都已经被泛型的创立者谨慎地创建并检查，所以我们可以研究这些 API 学习到很多关于泛型与 generic method 的良好设计经验。

`java.util` 的泛型相当简单，大部分都是 collection 类以及用来表示 collection 的元素类型的类型变量。在 `java.lang` 里有几个重要的泛型是比较困难的。它们不是 collection，而且也无法直接地看出为什么它们已经被改为泛型。研究这些困难的泛型会让我们更深入地了解泛型是如何运行，同时也会引入未涵盖在本书里的一些概念。明确地说，我们将要查看 Comparable 接口与 Enum 类（enumerated 类型的超类型，会在本章的稍后说明），而且会学到关于泛型的一个非常重要但却很少使用到的一个功能，也就是所谓的下限通配符（lower-bounded wildcard）。

在 Java 5.0 中，Comparable 接口已经被改为泛型了，且该类型变量详细地指出了什么样的类可以拿来作比较。大部分的类都会在其自身中去实现 Comparable。考虑 Integer：

```
public final class Integer extends Number implements Comparable<Integer>
```

从类型安全性的观点来看，原始的 Comparable 接口是有问题的。拥有两个无法彼此作有意义比较的 Comparable 对象是有可能的。在 Java 5.0 之前，非泛型的 Comparable 接口很有用，但并不完全符合要求。然而，此接口的泛型版本可以精确地撷取我们想要的信息：它告诉我们类型是否是兼容的，而且也告诉我们可以用它作什么比较。

现在来看一下 Comparable 类的子类。Integer 被声明为 final 而且不可以有子类，所以我们来看 `java.math.BigInteger`：

```
public class BigInteger extends Number implements Comparable<BigInteger>
```

如果我们实现 BigInteger 的子类 BiggerInteger，它从超类继承了 Comparable 接口。但请注意，它继承的是 `Comparable<BigInteger>` 而不是 `Comparable<BiggerInteger>`。这就表示 BigInteger 与 BiggerInteger 两对象是可以互相作比较的，这通常是一件好事。BiggerInteger 可以覆盖其超类的 `compareTo()`

method, 但它并不允许实现不同参数状态的 Comparable。也就是, BigInteger 无法同时继承 BigInteger 类与实现 Comparable<BigInteger> 接口 (一般来说, 类不允许实现相同接口的两个不同的参数化, 例如, 我们无法定义同时实现了 Comparable<Integer> 与 Comparable<String> 的类型)。

当你处理可比较的对象时 (例如在写排序算法时), 要记得两件事。第一, 把 Comparable 当作原始类型 (raw type) 是不够的, 对类型安全性来说, 你必须明确地指定它可以和什么东西作比较。第二, 类型并不是只能和它们自己作比较, 有时候它们也会和它们的超类作比较。为了更具体一些, 考虑 java.util.Collections.max() method:

```
public static <T extends Comparable<? super T>> T max(Collection<? extends T> c)
```

这是一个既长又复杂的 generic method 签名。我们来看一下:

- method 有个我们会在稍后返回的具有复杂界限的类型变量 T
- method 返回一个类型 T 的值
- method 的名称为 max()
- method 的自变量是 Collection。collection 的元素类型使用了有限制的通配符来表示, 我们并不知道该 collection 元素的确切类型, 但我们知道它们有一个 T 的上限条件, 也就是所有 collection 的元素都是类型 T 或 T 的子类。因此, collection 内的任何元素都可以用来当作 method 的返回值。

这样就非常容易理解了。我们在这节的其他地方看到了有上限条件的通配符。现在, 我们再来看一下 max() method 所使用的类型变量声明:

```
<T extends Comparable<? super T>>
```

首先, 这是在说类型 T 必须要实现 Comparable (对所有类型变量界限来说, 不管是类还是接口, 泛型语法都使用了 extends 关键字)。这是预期中的, 因为此 method 的目的是要在 collection 里寻找“最大的”对象。但请看一下 Comparable 接口的参数化。这是个通配符, 但它使用 super 关键字来作限制, 而不是 extends 关键字。这是个下限型通配符。? extends T 是我们所熟悉的上限条件, 它代表 T 或子类。? super T 就比较少用到, 它是指 T 或超类。

总的来说, 类型变量声明“T 是可和它自己或其父类作比较的类型”。Collections.min() 与 Collections.binarySearch() method 也有相似的签名。

对于其他的下限型通配符 (与 Comparable 无关) 的例子, 要考虑到 Collections 的 addAll(), copy() 与 fill() method。以下是 addAll() 的签名:

```
public static <T> boolean addAll(Collection<? super T> c, T... a)
```

这是个 varargs method, 该 method 会接受任意数量类型为 T 的自变量, 而且会把它们作为名为 a 的 T[] 来传递。它会将所有 a 的元素加到 collection c 中。该 collection 的元素类型是未知的, 但它有下限条件: 所有的元素都属于类型 T 或 T 的超类。不管是哪种类型, 都可以保证该数组的元素就是该类型的实例, 所以将这些数组元素加入到 collection 中是合法的。

应记住我们稍早讨论的上限型通配符。如果你有一个 collection 且该 collection 的元素类型就是上限型通配符, 那它就是非常有效率地只读。考虑 List<? Extends Serializable>。我们知道它所有的元素都是 Serializable, 所以像 get() 这样的 method 就会返回类型为 Serializable 的值。编译器不会让我们去调用像 add() 这样的 method, 因为列表的实际元素类型是未知的。你也不可以在列表上加入任意的可序列化对象, 因为它们实现的类也许不会是正确的类型。

由于上限型通配符会导致只读的 collection, 你可能会期望下限型通配符会产生只写的 collection, 然而事实并非如此。假设我们有个 List<? super Integer>, 在元素类型是未知的情况下, 唯一有可能的就是 Integer 或它的父类 Number 与 Object。不管实际类型是什么, 将 Integer 对象 (而不是 Number 或 Object 对象) 加入列表中是安全的, 而且不管实际元素类型是什么, 列表里所有的元素都是 Object 的实例。所以在这种情况下, 像 get() 这样的 List method 就会返回 Object。

最后, 我们将注意力转到 java.lang.Enum 类来。Enum 为所有的枚举类型 (稍后会做说明) 提供了有用的超类型。它实现了 Comparable 接口, 但却有一个令人感到困惑的泛型签名:

```
public class Enum<E extends Enum<E>> implements Comparable<E>, Serializable
```

在看第一眼时, 类型变量 E 的声明是循环的。我们再看仔细一点: 这个签名实际上是在说, Enum 必须要由本身为 Enum 的类型来参数化。如果我们查看签名的 implements 子句, 这个看似循环的类型变量声明的原因就会变得很明显。就如我们所看到的, Comparable 类通常会被定义为能与它们自己作比较, 而这些类的子类能与它们的超类比较。另一方面, Enum 并不是为它自己实现 Comparable 接口, 而是为它的子类 E。

## 枚举类型

在前面的章节中, 我们已经看到了 class 关键字是用来定义类类型, 而 interface 关键字则是用来定义接口类型。此节将要介绍 enum 关键字, 它是用来定义枚举类型 (非正式的说法是 enum)。枚举类型是 Java 5.0 中新加入的类型, 在这里要介绍的功能在之前的版本是无法使用的 (虽然有些可以被模仿出来)。



我们从基础开始：如何定义与使用枚举类型，包括一般程序设计上使用枚举类型与值的习惯用法。接下来，我们将会讨论更高级的 enum 特性以及说明在 Java 5.0 以前要如何模仿 enum。

## 枚举类型的基础

枚举类型是个具有可能值的有限（通常都很小）集合的引用类型，集合中的每一个都是单独被列出或枚举。这儿有个定义于 Java 中的简单枚举类型：

```
public enum DownloadStatus { CONNECTING, READING, DONE, ERROR }
```

就像类与接口一样，enum 关键字定义了一个新的引用类型。以上的单行 Java 程序代码定义了一个名称为 DownloadStatus 的枚举类型，此类型的主体是用逗号将类型的四个值分隔开的列表。这些值就像 static final 字段（这就是为什么它们的名称都要用大写来表示）一样，你可以用像 DownloadStatus.CONNECTING、DownloadStatus.READING 这样的名称来引用。DownloadStatus 类型的变量可以被指定为这四个值的其中一个或 null，但不能是其他值。枚举类型的值被称为枚举值（enumerated value），有时候也被称为枚举常量（enum constant）。

要定义比这还要更复杂的枚举类型是有可能的，我们会在本章的稍后介绍完整的 enum 语法。不过，现在你可以使用基本的语法来定义简单但非常有用的枚举类型。

## 枚举类型就是类

在介绍 Java 5.0 中的枚举类型之前，DownloadStatus 的值或许已在类或接口里使用了以下程序代码来变成整数常量：

```
public static final int CONNECTING = 1;
public static final int READING = 2;
public static final int DONE = 3;
public static final int ERROR = 4;
```

使用整数常量会有一些缺点，最重要的就是它缺少了类型安全性。例如，如果有个 method 期望下载状态是常量值，没有错误检查来防止我传送不合法的值。当我要使用 DownloadStatus.DONE 时，编译器无法告诉我我已经使用了常量 UploadStatus.DONE。

很幸运的是在 Java 里的枚举类型并不是单纯的整数常量。使用 enum 关键字所定义的类型实际上就是一个类，而且它的枚举值就是该类的实例。它提供了类型安全性：如果我尝试将 DownloadStatus 值传递给预期有 UploadStatus 的 method 时，编译器就会提出错误。枚举类型没有公共构造函数，所以程序无法创建一个新的未定义类型的实例。

如果该 `method` 期望的是一个 `DownloadStatus` 值，我们就可以确信将不会有一些未知类型的实例被传入。

如果你在写程序时已经习惯于使用整数常量来代替枚举类型，对枚举类型来说，你或许已经在整数上获得了许多实际的优点。然而，请保持你的看法：接下来的章节会说明一般的枚举类型程序设计的习惯用法，以及证明任何可以用整数常量做的事，`enums` 也可以更巧妙、更有效率且更安全地做到。不过，我们先来看看所有枚举类型最基本的特性。

## 枚举类型的特性

以下的列表描述了关于枚举类型的基本事实。这些都是你必须知道的 `enum` 特性，以了解及更有效地使用它们：

- 枚举类型没有公共构造函数，枚举类型唯一的实例就是这些枚举声明。
- 枚举类型不是 `Cloneable`，所以无法为现有的实例创建副本。
- 枚举类型实现了 `java.io.Serializable`，所以它们可以被序列化，但 Java 的序列化机制会特别处理它们，以确保不会有新的实例被创建。
- 枚举类型的实例是永远都不会变的：每一个枚举类型值都会保留它自己的特性（我们将在本章稍后的章节看到，你可以在枚举类型里加上你自己的字段与 `method`，这代表你可以创建具有可变部分的枚举值。但这并不建议使用，虽然它不会影响每一个值的基本特性）。
- 枚举类型的实例是存储于类型本身的 `public static final` 字段里。因为这些字段为 `final`，所以它们不可能被不恰当的值改写，例如，你无法将 `DownloadStatus.ERROR` 值指定给 `DownloadStatus.DONE` 字段。
- 依照惯例，枚举类型的值都是使用大写字母来编写，就和其他 `static final` 字段一样。
- 因为有一个严格限制的具有不同枚举值的集合，所以使用 `==` 运算符比较枚举值来代替调用 `equals()` `method` 一定是安全的。
- 然而，枚举类型的确有个可行的 `equals()` `method`。该 `method` 在内部使用 `==` 而且它被定义为 `final`，所以它无法被覆盖。这个可行的 `equals()` `method` 允许枚举值被用作类似 `Set`、`List` 与 `Map` `collection` 的成员。
- 枚举类型有个可行的 `hashCode()` `method`，与它们的 `equals()` `method` 一致。和 `equals()` 一样，`hashCode()` 也是被定义为 `final`。它允许枚举值配合类似 `java.util.HashMap` 这样的类一起使用。

- 枚举类型实现了 `java.lang.Comparable`，而且 `compareTo()` method 会依它们出现在 `enum` 声明里的顺序来排序枚举值。
- 枚举类型包含了可行的 `toString()` method，它会返回枚举值的名称。例如，`DownloadStatus.DONE.toString()` 在默认的情况下会返回“DONE”字符串。此 method 并没有被定义为 `final`，`enum` 类型可以依选择提供 `custom implementation`。
- 枚举类型提供了一个静态 `valueOf()` method，它与默认的 `toString()` method 相反。例如，`DownloadStatus.valueOf("DONE")` 会返回 `DownloadStatus.DONE`。
- 枚举类型定义了一个名为 `ordinal()` 的 `final` 实例 method，该 method 会为每一个枚举值返回一个整数。枚举值的顺序代表了它在 `enum` 声明中的值名列表中的位置（从零开始）。你通常不需要使用 `ordinal()` method，但它会被一些与枚举相关的工具所使用，这在本章的稍后会做介绍。
- 每个枚举类型都会定义一个名为 `values()` 的静态 method，它会返回由那个类型的枚举值所组成的数组。此数组包含了完整的值集合，依它们被声明的顺序，这对于迭代整个可能值的集合很有用。因为数组是可变的，`values()` method 一定会返回最近被创建与初始化的数组。
- 枚举类型是 `java.lang.Enum` 的子类，它是在 Java 5.0 中新出现的（`Enum` 本身并不是枚举类型）。你不能以手动方式扩展 `Enum` 类以产生枚举类型，如果试图这么做，就会产生编译错误。定义枚举类型的唯一方式就是使用 `enum` 关键字。
- 要扩展枚举类型是不可能的。枚举类型实际上是 `final`，但 `final` 关键字在它们的声明里是不必要且不被允许的。因为 `enum` 实际上是 `final`，所以它们不会是 `abstract`（在本章稍后将回头讨论这点）。
- 和类一样，枚举类型可以实现接口（我们在本章的稍后会看到枚举类型是如何定义 `method` 的）。

## 使用枚举类型

以下的章节说明了处理枚举类型的常见习惯用法。它们说明了 `switch` 语句配合枚举值的用法并介绍了重要的新 `EnumSet` 与 `EnumMap` collection。

### enum 与 switch 语句

在 Java 1.4 及之前的版本里，`switch` 语句只可以与 `int`、`short`、`char` 与 `byte` 值一起运作。因为枚举类型是值的有限集合，所以在概念上很适合与 `switch` 语句一起使用，而且这个语句在 Java 5.0 里用来支持枚举类型的使用。如果 `switch` 表达式在编译时的



类型为枚举类型，则 `cast` 标签必须是该类型的实例的完整名称。以下的程序代码显示了使用 `DownloadStatus` 枚举类型的 `switch` 语句。

```
DownloadStatus status = imageLoader.getStatus();
switch(status) {
case CONNECTING:
    imageLoader.waitForConnection();
    imageLoader.startReading();
    break;
case READING:
    break;
case DONE:
    return imageLoader.getImage();
case ERROR:
    throw new IOException(imageLoader.getError());
}
```

请注意，`case` 标签只是常量名称：`switch` 语句的语法不允许类名 `DownloadStatus` 在这里出现。忽略类名的功能是非常方便的，因为它会以其他方法出现在每个单独的 `case` 里。然而，省略掉类名的需求是令人感到讶异的，因为（在没有 `import static` 声明的情况下）类名在其他的相关环境中是必要的。

如果 `switch` 表达式（上述程序代码里的 `status`）被判定为 `null`，`NullPointerException` 就会被抛出。使用 `null` 作为 `case` 标签的值是不合法的。

如果你在枚举类型上使用 `switch` 语句而且没有包含 `default`，则对于标签或针对每个枚举值的 `case` 标签，编译器将很有可能产生 `-Xlint` 警告信息，好让你知道你没有编写程序代码来处理枚举类型所有可能的值（注5）。即使当你真的为每个枚举值写一个 `case` 时，还是要将 `default` 子句包含进来。这可以处理在你的 `switch` 语句编译完成之后，有新的值被加入枚举类型的情况。例如，以下的 `default` 子句可以加入之前的 `switch` 语句中：

```
default: throw new AssertionError("Unexpected enumerated value: " + status);
```

## EnumMap

当使用整数常量来代替枚举值时，常见的程序设计技术是将这些常量当作数组的索引。例如，如果 `DownloadStatus` 值被定义为 0 到 3 之间的整数，则我们可以将程序代码写成这样：

```
String[] statusLineMessages = new String[] {
    "Connecting...", // CONNECTING
```

注5： 在撰写这段时，预计这个警告信息在 Java 5.1 里会出现。

```
"Loading...",          // READING
"Done.",              // DONE
"Download Failed."    // ERROR
};

int status = getStatus();
String message = statusLineMessages[status];
```

这个技巧创建了从枚举整数常量到字符串的映射。我们不可以将Java的枚举值当成数组索引，但我们可以将它们当成是 `java.util.Map` 里的键。因为通常都会这么做，所以Java 5.0 定义了一个新的 `java.util.EnumMap` 类，用来完全针对此状况以优化它。`EnumMap` 需要一个枚举类型来作为它的键，而且所有可能值的数量是有限的，它使用了数组来保存所有可能的值。这样的实现意味着 `EnumMap` 比 `HashMap` 更有效率。以下就是和前面的程序代码相等的 `EnumMap` 程序代码：

```
EnumMap<DownloadStatus,String> messages =
    new EnumMap<DownloadStatus,String>(DownloadStatus.class);
messages.put(DownloadStatus.CONNECTING, "Connecting...");
messages.put(DownloadStatus.READING,    "Loading...");
messages.put(DownloadStatus.DONE,       "Done.");
messages.put(DownloadStatus.ERROR,      "Download Failed.");

DownloadStatus status = getStatus();
String message = messages.get(status);
```

和Java 5.0里的其他 `collection` 类一样，`EnumMap` 是一个可以接受类型参数的泛型。

当你在运用定义于其他地方的 `enum` 时，使用 `EnumMap` 来结合值与每个枚举类型的实例是非常恰当的。如果你定义了自己的 `enum` 值，你就可以创建一些必要的结合作为 `enum` 定义的一部分。我们会在稍后看看要怎么做。

## EnumSet

当使用基于整数的常量来代替枚举类型时，另一个程序设计的习惯用法是把所有的常量定义为2的  $n$  次方，以便这些常量可以像整数中的位标记那样来简洁地表示。请看以下的标记，它描述了适用于美国风格的浓缩咖啡饮品的选项：

```
public static final int SHORT      = 0x01;  // 8 盎司
public static final int TALL       = 0x02;  // 12 盎司
public static final int GRANDE     = 0x04;  // 16 盎司
public static final int DOUBLE     = 0x08;  // 2 小杯浓缩咖啡
public static final int SKINNY     = 0x10;  // 使用脱脂牛奶
public static final int WITH_ROOM  = 0x20;  // 加奶油
public static final int SPLIT_SHOT = 0x40;  // 半咖啡因
public static final int DECAF      = 0x80;  // 无咖啡因
```

这些2的  $n$  次方常量可以使用 `bitwise OR` 运算符(`|`)来创建易于运用的简单常量集合：

```
int drinkflags = DOUBLE | SHORT | WITH_ROOM;
```

bitwise AND 运算符 (&) 可以用来测试位是否存在:

```
boolean isBig = (drinkflags & (TALL | GRANDE)) != 0;
```

我们从这些使用位标记表示的二进制值以及操作它们的布尔运算符,就可以看到整数位标记仅只是简单的值集合。对于引用类型,例如 Java 的枚举值,我们可以使用 `java.util.Set` 来代替。由于这是在使用枚举值时重要且常做的事,Java 5.0 提供了特殊用途的 `java.util.EnumSet` 类。和 `EnumMap` 一样, `EnumSet` 能对枚举类型进行优化。它的成员必须是属于相同枚举类型的值,并且使用以位标记为基础的集合及简洁且快速的表达式,那些标记对应至每个枚举值的 `ordinal()`。

以上的浓缩咖啡程序代码可以使用 `enum` 与 `EnumSet` 来改写成如下的样子:

```
public enum DrinkFlags {
    SHORT, TALL, GRANDE, DOUBLE, SKINNY, WITH_ROOM, SPLIT_SHOT, DECAF
}

EnumSet<DrinkFlags> drinkflags =
    EnumSet.of(DrinkFlags.DOUBLE, DrinkFlags.SHORT, DrinkFlags.WITH_ROOM);

boolean isbig =
    drinkflags.contains(DrinkFlags.TALL) ||
    drinkflags.contains(DrinkFlags.GRANDE);
```

请注意,以上的程序代码可以使用单纯的静态导入让基于整数的代码尽可能简洁:

```
// 导入所有 static DrinkFlag 枚举常量
import static com.davidflanagan.coffee.DrinkFlags.*;
```

对于 `import static` 声明的细节,请参阅第二章的“包与 Java 命名空间”。`EnumSet` 为枚举值的初始化定义了一些有用的 factory method。以上所显示的 `of()` method 是重载 (overloaded) method, 该 method 有好几个版本,而且各自会接受固定数量的自变量。可以接受任意数量自变量的 `varargs` (请看第二章) 格式也已被定义。这里有一些使用 `of()` 以及相关 `EnumSet` factory 的其他方法:

```
// 让以下范例在排版上好看一点
import static com.davidflanagan.coffee.DrinkFlags.*;

// 我们可以从集合里移除个别成员或是一组成员
// 从包含了所有的枚举值的集合开始,移除一个子集合:
EnumSet<DrinkFlags> fullCaffeine = EnumSet.allOf(DrinkFlags.class);
fullCaffeine.removeAll(EnumSet.of(DECAF, SPLIT_SHOT));

// 这是另一个可以达到相同结果的技术
EnumSet<DrinkFlags> fullCaffeine =
    EnumSet.complementOf(EnumSet.of(DECAF, SPLIT_SHOT));
```



```
// 这是个你可能会需要的空集合
// 请注意，因为我们没有指定一个值，所以我们必须指定元素类型
EnumSet<DrinkFlags> plainDrink = EnumSet.noneOf(DrinkFlags.class);

// 你也可以轻易地描述值的连续子集合
EnumSet<DrinkFlags> drinkSizes = EnumSet.range(SHORT, GRANDE);

// EnumSet 是 Iterable，它的 iterator 会依 ordinal() 顺序返回值，
// 所以要逐一处理 EnumSet 的元素是很容易的
for(DrinkFlag size : drinkSizes) System.out.println(size);
```

在这里所显示的范例程序说明了 `EnumSet` 类的用法与功能。然而请注意，`EnumSet<DrinkFlags>` 并不是非常适合用来描述浓缩咖啡。饮品 `EnumSet<DrinkFlags>` 可能会被过度指定，例如包括 `SHORT` 与 `GRANDE`；它也可能被指定不足，如完全不包括饮料的大小。

最根本的问题就是 `DrinkFlag` 类型只是我们在本章开头所使用的整数位标记的简单转换而已。通过以下的接口可获得较好且较完整的表达方式，它需要来自五个不同枚举类型的值以及来自第六个 `enum` 的一组值。`enum` 在接口中被定义为嵌套类型（请参阅第三章）。此范例强调了由枚举类型所提供的类型安全性。例如，在指定饮料大小的地方是不可能指定饮料浓度的。

```
public interface Espresso {
    enum Drink { LATTE, MOCHA, AMERICANO, CAPPUCCINO, ESPRESSO }
    enum Size { SHORT, TALL, GRANDE }
    enum Strength { SINGLE, DOUBLE, TRIPLE, QUAD }
    enum Milk { SKINNY, ONE_PERCENT, TWO_PERCENT, WHOLE, SOY }
    enum Caffeine { REGULAR, SPLIT_SHOT, DECAF }
    enum Flags { WITH_ROOM, EXTRA_HOT, DRY }

    Drink getDrink();
    Size getSize();
    Strength getStrength();
    Milk getMilk();
    Caffeine getCaffeine();
    java.util.Set<Flags> getFlags();
}
```

## 高级的 enum 语法

到目前为止所显示的范例使用的都是最简单的 `enum` 语法，其中 `enum` 的主体只是由逗号区隔开的值名列表所构成。完整的 `enum` 语法实际提供了更多的功能和灵活性：

- 你可以为枚举类型定义你自己的字段、`method` 与构造函数。
- 如果你定义了一个或多个构造函数，就可以通过将构造函数自变量放在括号内并接在值名后面来调用每个枚举值的构造函数。

- 虽然 enum 不可以扩展任何东西，但它可以实现一个或多个接口。
- 最难理解的是个别枚举值可以有它们自己的类主体，这个类主体覆盖了该类型所定义的 method。

我们不正式说明针对以上这些高级 enum 声明的语法，而是在以下的范例里说明。

### 枚举类型的类主体

请看以下所定义的类型 Prefix。它是 enum，且在枚举值列表后包含了一般的类主体。它定义了两个实例字段以及针对这些字段的访问 method。它定义了构造函数，这会初始化实例字段。每个具名枚举类型值后面接着被放在括号中的构造函数自变量。

```
public enum Prefix {  
    // 这些是这个枚举类型的值  
    // 每一个值后面都接着被放在括号中的构造函数自变量  
    // 每一个值之间是用逗号分隔，而且这个值的列表是由一个分号结束，  
    // 以与后面接着的类主体分开  
    MILLI("m", .001),  
    CENTI("c", .01),  
    DECI("d", .1),  
    DECA("D", 10.0),  
    HECTA("h", 100.0),  
    KILO("k", 1000.0); // 请注意分号  
  
    // 这是会被以上每个值调用的构造函数  
    Prefix(String abbrev, double multiplier) {  
        this.abbrev = abbrev;  
        this.multiplier = multiplier;  
    }  
  
    // 这些是由构造函数设定的专用字段  
    private String abbrev;  
    private double multiplier;  
  
    // 这些是针对专用字段的访问 method，  
    // 它们是每个枚举类型的值的实例 method  
    public String abbrev() { return abbrev; }  
    public double multiplier() { return multiplier; }  
}
```

请注意，如果值后面接着类主体，则 enum 语法在最后一个枚举值的后面就需要有分号。在没有类主体的情况下，这个分号可以被省略。还有要注意的是，enum 语法允许在最后一个枚举值后面加上逗号。在尾端的逗号看起来有点奇怪，但如果将来你增加了新的枚举值或将已存在的值重新排列时，这样就可以防止语法错误。

## 实现接口

enum 不能被声明以扩展类或枚举类型，但是，枚举类型实现一个或多个接口是完全合法的。例如，假设你用像 Prefix 所具有的 abbrev() method 来定义新的枚举类型 Unit。在这样的情况下，你可以为任何有缩写词的对象定义一个 Abbrevable 接口。你的程序代码看起来会像这样：

```
public interface Abbrevable {
    String abbrev();
}

public enum Prefix implements Abbrevable {
    // 此 enum 类型的主体和上面的一样
}
```

## 值所特有的类主体

除了为枚举类型本身定义类主体之外，你也可以为该类型内的个别枚举值提供类主体。我们已经看到可以为枚举类型增加字段，并且用构造函数来初始化这些字段，这给了我们值所特有的数据 (value-specific data)。为每一个枚举值定义一个类主体的功能表示我们可以为每个枚举值编写 method，这给了我们值所特有的行为 (value-specific behavior)。在定义枚举类型来表示表达式中的运算符或某些虚拟机中的操作码 (opcode) 时，值所特有的行为很有用。例如，Operator.ADD 常量可能会有 compute() method，该 method 的行为与 Operator.SUBTRACT 常量的不同。

如果要为个别的枚举值定义类主体，只要在值名与其构造函数自变量的后面接着大括号括住的类主体即可。每个值仍然需要用逗号来彼此分开，而列表里的最后一个值必须使用分号来与类型的类主体分开。在针对类与 method 主体的大括号出现的情况下，这个必要的标点符号很容易就会被遗忘。

你所编写的每个值所特有的类主体会造成枚举类型的匿名子类的创建，而且会让枚举值成为那个匿名子类的单独实例 (singleton instance) (枚举类型无法被扩展，但它们并不是像 final 类中那样严格的 final，因为它们可以拥有这些匿名子类)。因为这些子类是匿名的，所以你无法在程序代码里引用它们：每个枚举值在编译时的类型都是枚举类型，而不是那个值所特有的匿名子类。因此，在值所特有的类主体中，你唯一可以做的事就是覆盖由类型本身所定义的 method。如果你定义了新的公共字段或 method，就不能引用或调用它 (当然，从覆盖的 method 定义你要调用或使用的辅助 (helper) method 或字段是完全合法的)。

常见的模式是在类型特有的类主体的 method 里定义默认的行为。然后，每个需要与默认行为不同的行为的枚举值，就可以覆盖其值所特有的类主体内的 method。这个模式有



个很有用的变体，就是将特定类型的类主体的method声明为abstract，并为每个枚举值的method定义值所特有的实现。如果类型特有的method为abstract，编译器就会强迫你为类型里的每个枚举值实现那个method，要不小省略这个实现是不可能的。请注意，即使类型特有的类主体包含了abstract method，但枚举类型的整体也并不是abstract（而且也可能不会被声明为abstract），因为每个值所特有的类主体实现了method。

以下的程序代码是大型范例的摘录，它使用枚举类型来代表基于堆栈的仿真CPU的操作码。Opcode枚举类型定义了一个abstract method perform()，它其后由类型的各个值的类主体所实现。此类型包含了一个构造函数来说明每个枚举值的完整语法：名称、构造函数自变量以及类主体。enum语法要求枚举值以及它的类主体要先出现。然而，如果略掉这些值先查看类型特有的类主体时，程序代码就会非常容易了解：

```
// 这些是我们的堆栈机器可以执行的操作码
public enum Opcode {
    // 将单一操作数推入堆栈
    PUSH(1) {
        public void perform(StackMachine machine, int[] operands) {
            machine.push(operands[0]);
        }
    }, // 要记得使用逗号来分隔枚举值

    // 将堆栈最上面的两个值加起来并推出结果
    ADD(0) {
        public void perform(StackMachine machine, int[] operands) {
            machine.push(machine.pop() + machine.pop());
        }
    },

    /* 为了让程序简单点，其他的opcode值已被省略 */

    // 如果等于零就进入分支
    BEZ(1) {
        public void perform(StackMachine machine, int[] operands) {
            if (machine.top() == 0) machine.setPC(operands[0]);
        }
    }; // 请记住在类主体之前要有分号

    // 这是类型的构造函数
    Opcode(int numOperands) { this.numOperands = numOperands; }

    int numOperands; // 它预期会有多少个整数操作数？

    // 每个opcode常量都必须在值所特有的类主体中实现
    // 此abstract method用来执行它所代表的操作
    public abstract void perform(StackMachine machine, int[] operands);
}
```

## 值所特有的类主体的使用时机

当每个枚举值都必须执行唯一的某种运算时,值所特有的类主体就会有非常强大的语言特性。但请记住,值所特有的类主体是高级的特性,它不常被使用,而且对于经验较少的程序员可能会造成混淆。在你决定要使用这个特性之前,请先确定那是必要的。

在使用值所特有的类主体之前,要确定你的设计对于此特性不会太简单也不会太复杂。首先,确认你真的需要值所特有的行为,而不是单纯的值所特有的数据。值所特有的数据可以被编码于构造函数自变量里,就像稍早的 Prefix 范例所显示的。例如,把那个范例改写为使用值所特有的 `abbrev() method` 版本是不必要也不恰当的。

接下来,想一下枚举类型是否充分符合你的需要。如果你的设计需要具有复杂实现的值所特有的 `method` 或每个值需要很多 `method` 时,你会发现把所有东西都写在单一类型中是很繁琐的。我们可以考虑使用传统的 `class` 与 `interface` 声明来声明你自己的自定义类型层次以及任何你需要的单独实例。

如果值所特有的行为在枚举类型的结构里真的需要,那么值所特有的类主体就会适合。值所特有的主体是很精致还是会令人混淆,这是见仁见智的问题,有些程序员会倾向于尽可能避免使用。有个对一些具有吸引力的替代方案,就是在使用 `switch` 语句来将每个值作为个别 `case` 的类型特有的 `method` 中编码值特有的行为。以下的 `enum` 里的 `compute() method` 就是一个例子。此枚举类型的单纯性使得 `switch` 语句成为值所特有的类主体的替代方案:

```
public enum ArithmeticOperator {
    // 枚举值
    ADD, SUBTRACT, MULTIPLY, DIVIDE;

    // 使用了 switch 语句的值所特有的行为
    public double compute(double x, double y) {
        switch(this) {
            case ADD:      return x + y;
            case SUBTRACT: return x - y;
            case MULTIPLY: return x * y;
            case DIVIDE:   return x / y;
            default: throw new AssertionError(this);
        }
    }

    // 使用此 enum 的测试案例
    public static void main(String args[]) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        for(ArithmeticOperator op : ArithmeticOperator.values())
            System.out.printf("%f %s %f = %f\n", x, op, y, op.compute(x,y));
    }
}
```

使用 `switch` 语句的方法有个缺点，就是每次你增加新的枚举值时，就必须记得在 `switch` 语句里加入相对应的 `case` 语句。而且如果有一个以上的 `method` 使用了 `switch` 语句，你就必须同时维护它们的 `switch` 语句。如果忘记使用 `switch` 语句来实现值所特有的行为，就会导致运行时的 `AssertionError`。在值所特有的类主体覆盖类型特有的类主体内的 `abstract method` 时，同样的省略会造成编译错误，但可以被尽快修正。

值所特有的 `method` 和类型特有的 `method` 中的 `switch` 语句在性能上相当类似。在一个状况中的虚拟 `method` 调用的耗费，会与在另一个状况中 `switch` 语句的耗费平衡。值所特有的类主体会造成额外类的产生，就存储空间与加载时间来看，这些都会成为额外的负担。

### enum 类型上的限制

Java 对于能在枚举类型里出现的程序代码作了一些限制。实际上，你不会常常遇到这些限制，但你还是应该要知道。

当你定义一个枚举类型时，编译器在私下做了很多的事情：它会创建一个扩展了 `java.lang.Enum` 的类，并且产生 `values()` 与 `valueOf()` `method` 以及保存枚举值的静态字段。如果你为类型加上类主体，就不应该加上名称会与自动产生的成员名称冲突或是与继承自 `Enum` 的 `final method` 冲突的成员。

`enum` 类型不可以被声明为 `final`。枚举类型是一个有效的 `final`，而且编译器不允许你去扩展 `enum`。但是如果 `enum` 包含了值所特有的类主体，那么针对 `enum` 所产生的类文件在技术上就不会被声明为 `final`。

Java 里的类型不可以同时为 `final` 与 `abstract`。因为枚举类型为 `final`，它们不可以被声明为 `abstract`。如果 `enum` 声明的类型特有的类主体包含了 `abstract method`，编译器就会要求每个枚举值要有值所特有的类主体，其中包含了 `abstract method` 的实现。就单独的个体来看，以这个方式定义的枚举类型并不是 `abstract`。

构造函数（也就是实例字段的初始化程序）和枚举类型的实例初始化程序块受到了全面但模糊的限制：它们不可以使用类型的静态字段（包含枚举值本身）。这是因为枚举类型（以及所有的类型）的静态初始化的进行方式是由上而下。枚举值是出现在类型顶端的静态字段而且会先被初始化。因为它们都是自行输入（`self-typed`）的字段，所以它们会调用构造函数以及该类型的其他实例初始化程序代码。这代表实例初始化程序代码是在类的静态初始化完成之前被调用的。因为静态字段尚未被初始化，所以编译器并不允许它们被使用。唯一的例外是静态字段的值是编译器可解析的编译时常量表达式（例如 `integer` 与 `string`）。



如果你为枚举类型定义了构造函数，那么它不可以使用 `super()` 关键字来调用超类构造函数，这是因为编译器会自动地将隐藏的 `name` 与 `ordinal` 自变量插入你所定义的构造函数里。如果你为该类型定义了多个构造函数，那么使用 `this()` 来从一个构造函数调用另一个构造函数是可行的。请记住，个别的枚举值（如果你定义了）的类主体是匿名的，这代表它们根本不可能会有任何的构造函数。

## 具有类型安全性的 enum 模式

为了更深入地了解 `enum` 关键字的运作方式，或是要模拟 Java 5.0 之前的枚举类型，了解具有安全性的 `enum` 模式是非常有用的。此模式在 Joshua Block（注 6）的《Effective Java Programming Language Guide》（Addison Wesley）一书中有精彩的描述，在这里我们不会介绍太多。

如果你想要使用 Java 5.0 之前的枚举类型 `Prefix`（在本章稍早介绍过），可以使用和以下类似的类来模拟它。但请注意，这个类的实例将无法和 `switch` 语句或 `EnumSet` 与 `EnumMap` 类一起运行。此外，这里所显示的程序代码并没有包含编译器为真正的枚举类型自动产生的 `value()` 或 `valueOf()` method。像这样的类不会有像枚举类型所提供的特殊序列化（`serialization`）支持，所以如果你让它可序列化，就必须提供 `readResolve()` method 来预防创建枚举值的多个实例时的反序列化（`deserialization`）。

```
public final class Prefix
// 这些是自行输入的常量
public static final Prefix MILLI = new Prefix("m", .001);
public static final Prefix CENTI = new Prefix("c", .01);
public static final Prefix DECI = new Prefix("d", .1);
public static final Prefix DECA = new Prefix("D", 10.0);
public static final Prefix HECTA = new Prefix("h", 100.0);
public static final Prefix KILO = new Prefix("k", 1000.0);

// 让这些成为专用字段，以让实例永远都不会改变
private String name;
private double multiplier;

// 这是专用构造函数，所以除了上面的实例以外，没有实例会被创建
private Prefix(String name, double multiplier) {
    this.name = name;
    this.multiplier = multiplier;
}

// 这些访问 method 是公共的
public String toString() { return name; }
```

注 6: Josh 是 JSR 201 委员会里的联合主席，该委员会开发了 Java 5.0 里的许多新的语言特点。同时，他也是枚举类型的创造者与幕后推动者。

```
    public double getMultiplier() { return multiplier; }  
}
```

## 注释

注释 (annotation) 提供了一个将任意信息或元数据 (metatdata) 与程序元素结合在一起的方式。在语法上, 注释的使用就像修饰符一样, 可以用于包、类型、构造函数、method、字段、参数及局部变量的声明上。存储于注释中的信息采用了 name=value 对的格式, 其类型是由注释类型所指定的。注释类型是一种 interface, 它们也可以通过 Java Reflection API 来提供访问。

注释可被用来将任何你想要的信息与程序元素结合在一起。唯一的基本规则就是注释不会对程序的运行造成任何的影响: 即使你增加或移除注释, 程序代码还是一样地运行。换个方式说, 就是 Java 解释器会忽略注释 (虽然我们可以在运行时通过 Java Reflection API 来取得注释)。因为 Java VM 会忽略注释, 所以注释类型是没有用的, 除非它伴随着可以将信息存储于注释类型里的工具。本章我们将会介绍标准的注释与 meta-annotation 类型, 就像 Override 与 Target 一样。与这些类型相关的工具就是 Java 编译器, 它必须以特定方式来处理它们 (本节在稍后会做说明)。

我们可以很容易就猜想到, 可以使用很多个注释 (注 7)。一个局部变量可能会用名为 NonNull 的类型来做注释, 就像是在断言该变量不会有 null 值一样。相关 (假设的) 程序代码分析工具接着会解析程序代码而且会尝试验证该断言。JDK 包含了一个名为 apt (Annotation Processing Tool, 注释处理工具) 的工具, 它为注释处理工具提供了一个框架: 它会针对注释扫描该源代码并特别调用你所提供的已写好的注释处理器类 (annotation processor class)。更多关于 apt 的说明请参阅第八章。你将会发现注释在企业级的程序设计里是最常被用到的, 在那里它们可以取代像 XDoclet 这样的工具, 它用来处理被内嵌在任意 javadoc 注释里的元数据。

本节将从介绍与注释相关的术语开始。然后会介绍 Java 5.0 里的标准注释类型, 你可以立刻将 javac 所提供的注释使用在你的程序代码里。接下来, 我们会说明编写注释的语法, 同时还会简短地说明在运行时使用 Java Reflection API 来查询注释。之后, 我们要介绍较难以理解的部分 —— 定义新的注释类型, 不过很少有程序员会这么做。本章最后的部分则是讨论 meta-annotation。

---

注 7: 这样的使用方式很快就会有。在编写这段时, JSR 250 通过了 Java Community Process 的方式来为 J2SE 与 J2EE 定义一些标准的常用注释。

## 注释的概念与术语

注释的最主要的概念就是注释是将信息或元数据与程序元素结合在一起。注释并不会影响到Java程序的运行方式，但它们会影响到像是编译器的警告或辅助工具的行为，例如文件产生器、stub generator等。

在讨论注释时以下的术语经常会被用到。特别重要的是注释（annotation）与注释类型（annotation type）之间的差异。

### 注释

注释把信息或元数据与Java程序元素结合在一起。注释使用了在Java 5.0里所介绍的新的语法与运行方式，就像修饰符一样，例如public或final。每一个注释都有一个名字以及零或多个成员。每一个成员都有一个名称和一个值，而这些name=value对就带有注释的信息。

### 注释类型

注释的名称和被注释类型所定义的名称、类型以及它的成员的默认值一样。注释类型是一个Java接口，此接口对于它的成员有一些限制，同时在使用它的声明时也有一些新的语法。当你使用Java Reflection API来查询注释时，它的返回值是一个对象，该对象实现了注释类型的接口而且也允许查询各个注释成员。Java 5.0在java.lang包里包含了三个标准的注释类型，我们将在本章稍后的“使用标准注释”一节里看到这些注释。

### 注释成员

注释成员被声明在注释类型里就像无自变量的method一样。method名称与返回类型定义了该成员的名称与类型。特殊的default语法允许针对任一个注释成员声明默认值。注释会出现在含有name=value对的程序元素里，它为所有没有默认值的注释成员定义值，而且也包含了覆盖其他成员的默认值。

### 标志注释（marker annotation）

没有定义任何成员的注释类型就被称为标志注释（marker annotation）。这类注释所带有的信息就是它出现与否。

### meta-annotation

用来声明注释类型的注释就称为meta-annotation。Java 5.0在java.lang.annotation包里包含了几个标准的meta-annotation类型，它们被用来指定该注释所适用的程序元素。

### target

target类型的注释是程序元素的注释。此注释适用于包、类型（类、接口、枚举类型甚至是注释类型）、类型成员（method、构造函数、字段与枚举值）、method参



数以及局部变量（包括循环变量以及 catch 参数）。注释类型的声明可以包含一个 meta-annotation，它限制了该注释类型允许的目标。

#### *retention*

注释的 *retention* 说明包含在注释里的信息可以被保留多长的时间。有些注释会被编译器丢弃而且只会出现在源代码里，其余的都会被编译进类文件。这些被编译进类文件的注释中，有些会被虚拟机所忽略，而有些则会在包含注释的类被加载时被虚拟机所读取。注释类型的声明可以使用 meta-annotation 来为该类型的注释指定它的 retention。可被虚拟机加载的注释在运行时是可见的 (runtime-visible)，而且可以被 `java.lang.reflect` 的 reflective API 所查询。

#### 元数据

当讨论到注释时，元数据 (metadata) 这个术语通常是指注释所隐含的信息或该注释本身。因为这个术语在计算机程序设计书籍里被使用在很多不同的地方，所以在本章中会尽量避免使用它。

## 使用标准注释

Java 5.0 在 `java.lang` 包里定义了三个标准的注释类型。以下的小节就是说明这三个注释类型以及解释如何使用它们来注释你的程序代码。

### Override

`java.lang.Override` 是一个标志注释类型，可以用它来帮没有其他程序元素的 method 做注释。这种类型的注释是作为一个 assertion 提供的，这个有注释的 method 会覆盖超类的 method。如果你在 method 里使用了这个注释，且该 method 并没有覆盖其超类的 method，则编译器会对这种情形产生一个编译错误并告知你。

这个注释是针对一般的程序设计错误种类而设计的，这样的错误是当你试图覆盖超类 method 时发生 method 名称或签名错误。在这种情况下，你也许已经重载了 method 名称，但实际上并没有覆盖该 method，而你的程序代码并没被调用到。

要使用这个注释类型，只要在 method 放置修饰符的地方将 `@Override` 包含进来就可以了。依据惯例，`@Override` 出现在修饰符之前，而在 `@` 字符与 `Override` 名称之间不可以有空白，虽然这在技术上是允许的。请注意，因为 `java.lang` 包会自动地被导入，所以在使用这个注释类型时你并不需要将这个包包含在里面。这里有一个在 method 里使用 `@Override` 注释的范例，它并没有正确地覆盖其超类的 `toString()` method。

```
@Override
public String toString() {    // 请注意，这儿有拼错的词！
```

```
// 用方括号括住超类的输出
return "[" + super.toString() + "];
}
```

在没有注释的情况下,单词的拼写错误可能会被忽略,而我们也可能会遇到难以理解的错误:为什么 `toString()` method 不能正确运行?但因为有了注释,编译器给我们的答复是: `toString()` method 不会如预期般地运行,因为实际上它并没有被覆盖。

请注意, `@Override` 注释只可以应用于打算要覆盖其超类的 method 的 method,而不可用于实现定义于接口中的 method。如果你没有正确地实现接口的 method,编译器就会产生错误。

## Deprecated

`java.lang.Deprecated` 是一个标志注释,它和 `@deprecated` 这个 javadoc tag 非常的相似(请看第七章关于编写 Java 文件注释的更详细的介绍)。如果你使用 `@Deprecated` 来帮一个类型或类型成员做注释,那么它告诉编译器使用有注释的元素是不被允许的。如果你在程序代码中使用(或是扩展、覆盖) `deprecated` 类型或成员,而该类型或成员并不是它们自己所声明的 `@Deprecated`,则编译器会产生警告。

请注意, `@Deprecated` 注释类型并不会反对 `@deprecated` 这个 javadoc tag。`@Deprecated` 注释是为 Java 编译器而设计的。就另一方面而言, javadoc tag 是提供给 *javadoc* 工具使用且作为说明文件提供的,它也包含了为什么程序元素不被使用以及它被什么东西所代替或可以用什么东西来取代它等描述。

在 Java 5.0 里,编译器会继续去寻找 `@deprecated` 这个 javadoc tag,同时使用它们来产生警告信息。然而,这样的用法已经过时了,除了 `@deprecated` 之外,你应该开始使用 `@Deprecated` 注释。

这里有个同时使用了注释与 javadoc tag 的范例:

```
/**
 * Sony Betamax 录像带格式
 * @deprecated 这种格式的播放器已经没有人有了,取而代之的是 VHS。
 */
@Deprecated public class Betamax { ... }
```

## SuppressWarnings

`@SuppressWarnings` 注释会选择性地关闭类、method 或字段以及变量初始化程序的编

译器所产生的警告 (compiler warning) (注8)。在 Java 5.0 里, Sun 的 *javac* 编译器有一个很强大的 *-Xlint* 选项, 它会使得你的程序代码产生关于 “lint” 的警告, 这是合法的, 但比较像是要表示程序设计上的错误。例如, 当你使用一个没有指明类型参数值的泛型 *collection* 类时, 这些包含了 “未校验的警告 (unchecked warning)” 的警告就会出现; 或是在 *switch* 语句里有一个 *case* 没有使用 *break*、*return* 或 *throw* 来结束而且允许控制权落到下一个 *case* 时, 这样的警告也会出现。

一般来说, 当你从编译器里看到了这些 lint 警告时, 就应该要去调查产生这些警告的程序代码。如果真的是错误, 就必须加以更正。如果它是个很简略的程序, 你可以依你自己的习惯重新编写程序代码, 这样就不会再有警告了。例如, 如果该警告信息是在告诉你并没有将枚举类型所有可能的值都表示在 *switch* 语句的 *case* 子句里, 那么你可以在 *switch* 语句里加入具有保护作用的 *default case* 来防止警告的发生, 即使你很确定该 *default case* 不会被调用到。

另一方面, 有时候要避免这样的错误时你并不需要做任何事情。例如, 如果你在程序代码里使用了一个泛型 *collection* 类, 而该类必须与之前的非泛型的程序代码交互时, 你就无法避免这个未校验的警告信息的产生。这是 *@SuppressWarnings* 可以出现的地方: 将这个注释加到最接近的相关修饰符 (一般来说是 *method* 修饰符) 的地方, 以告诉编译器你意识到了这个问题, 且它应该停止在该问题上对你的纠缠。

和 *Override* 与 *Deprecated* 不一样, *SuppressWarnings* 并不是个标志注释。它有一个名为 *value* 且类型为 *String[]* 的单一成员。此成员的值是被隐藏的警告的名称。*SuppressWarnings* 注释并没有定义被允许的警告名称, 这对编译器实现而言是一个问题。对 *javac* 编译器来说, 可以被 *-Xlint* 选项所接受的警告名称对 *@SuppressWarnings* 注释来说也是合法的。而指明任何你所想要的警告名称也是合法的, 编译器会忽略 (但可能会有警告) 它们无法识别的警告名称。

所以, 为了防止名称为 *unchecked* 与 *fallthrough* 的警告发生, 你可以使用和下面一样的注释。注释的语法是在该注释类型的后面加上一个小括号, 并使用逗号来将 *name=value* 对隔开。在这种情况下, *SuppressWarnings* 注释类型只定义了单一成员, 所以在小括号内只有一对值。因为成员值是一个数组, 所以使用大括号来将数组元素括在里面:

```
@SuppressWarnings(value={"unchecked","fallthrough"})
public void lintTrap() { /* 省略了简略的method主体 */ }
```

注8: 在编写本章时, *javac* 编译器并没有支持 *@SuppressWarnings* 注释。此注释预计会在 Java 5.1 里得到支持。



我们可以将这个注释缩写。当注释只有一个单一成员 (single member) 且该成员的名称为 “value” 时, 你可以省略注释里的 “value=”。所以以上的注释可以被重写成:

```
@SuppressWarnings({"unchecked","fallthrough"})
```

希望你任何method里都不常遇到一些无法解决的lint警告, 而只需要去防止一个单一名称的警告的发生。在这种情况下, 还有另一种注释缩写式。当在写一个只包含单一成员的数组值时, 你可以将大括号省略。在这种情况下, 我们有像这样的注释写法:

```
@SuppressWarnings("unchecked")
```

## 注释语法

在标准注释类型的说明里, 我们已经看过marker annotation与single-member annotation的语法了, 还包括了当single member的名称为“value”时的缩写以及当数组类型(array-typed) 成员只有单一数组元素时的缩写法。这节将会说明 annotation 的完整语法。

注释的组成是在@字符后加上注释类型的名称(可以包含一个包的名称)以及紧跟在之后的一对小括号, 同时在小括号里将该注释类型所定义的每一个成员用逗号将name=value对的列表分隔开。如果该注释类型为该成员定义了默认值, 则这些成员可能会出现在任何序列里或被忽略。每个value都必须是直接量或编译时常量、嵌套注释或数组。

在本章的结尾处, 我们定义了一个名为Reviews的注释类型, 该类型有一个单一的成员, 且该成员是一个@Review注释的数组。此Review注释类型有三个成员: “reviewer”是个字符串, “comment”是个有默认值且非必要的字符串, 而“grade”是个嵌套枚举类型Review.Grade的一个值。假设Reviews与Review类型都被正确地导入, 则使用这些类型的注释看起来就会像这样(请注意这个注释里使用了嵌套注释、枚举类型以及数组):

```
@Reviews({ // 只有一个值的注释, 所以在这里“value=”被省略了
    @Review(grade=Review.Grade.EXCELLENT,
            reviewer="df"),
    @Review(grade=Review.Grade.UNSATISFACTORY,
            reviewer="eg",
            comment="This method needs an @Override annotation")
})
```

另一个很重要的注释语法规则是没有程序元素会有一个以上的相同注释的实例。这是不合法的, 例如, 在类里放入多个@Review注释。这就是为什么@Review注释被定义为允许@Review注释的数组。

## 注释成员的类型与值

注释成员的值必须是非null的编译时常量表达式，它必须与成员的声明类型兼容。被允许的成员类型有基本类型、String、Class、枚举类型、注释类型以及以上任何类型的数组（但不可以是数组的数组）。例如，表达式`2*Math.PI`以及`"hello"+"world"`是合法的值，且成员的类型分别为double与String。

在本章结尾处我们定义了一个名为UncheckedExceptions的注释类型，它唯一的成员是一个扩展自 RuntimeException的类的数组。此类型的注释看起来会像这样：

```
@UncheckedExceptions({
    IllegalArgumentException.class, StringIndexOutOfBoundsException.class
})
```

## 注释目标

注释最常被置于类型定义（例如类）以及它们的成员里面（例如method与字段），也可以出现在包、参数以及局部变量里。本节为这些较不常见的注释目标提供了较多的信息。

包的注释（package annotation）出现在名为package-info.java的文件里的package声明之前。这个文件不应该包含任何类型的声明（“package-info”并不是一个合法的Java标识符，所以它不能包含任何公共的类型定义），它应该包含选择性的javadoc comment、零个或多个注释以及package声明。例如：

```
/**
 * 此包拥有我们自定义的注释类型
 */
@com.davidflanigan.annotations.Author("David Flanagan")
package com.davidflanigan.annotations;
```

当package-info.java文件被编译后，它会产生一个名为package-info.class的类文件，该文件包含了一个虚构的接口声明。此接口没有成员且它的名称package-info并不是一个合法的Java标识符，所以它无法被使用于Java源代码里。它的存在对有类或运行时保留的包注释而言就像是一个占位符一样。

请注意，包注释出现在任何package或import声明的作用范围之外。这意味着该包注释应该包含该注释类型（除非该包是java.lang）的包名。

在method参数、catch子句参数以及局部变量中出现的注释，对这些程序元素来说就像是修饰符列表的一部分。Java类文件的格式并没有为在局部变量或catch子句参数上存储注释来做准备，所以这些注释都会被保留在源代码里。然而，method参数的注释可以被保留在类文件里且会拥有类或运行时保留。

最后, 请注意枚举类型定义的语法不允许为枚举值指定任何的修饰符。然而, 它允许任何的值有注释。

## 注释与默认

注释必须为每一个被注释类型所定义且没有默认值的成员赋一个值。当然, 注释也可以和其他的成员一样将值置于注释里面。

了解默认值是如何被处理的是一个很重要的细节。默认值会被存储于该注释类型的类文件里, 而它并没有被编译进注释。如果你修改了注释类型以至于它的成员改变了默认值, 则这个改变将会影响所有没有为该成员指定明确值的注释类型。而已被编译过的注释也会受到影响, 即使在改变该类型后它们并没有被重新编译过。

## 注释与 Reflection

在 Java 5.0 里, `java.lang.reflect` 的 Reflection API 已经被扩展来支持运行时可见的注释 (如果它的注释类型被指定为 `runtime retention`, 那么注释只有在运行时是可见的, 也就是说, 注释被存储于类文件里, 所以当类文件被加载时, 注释可以被 Java VM 所读取)。本节会简短地说明新的反射能力。对注释来说, `AnnotatedElement` 表示一个可以被查询的程序元素。它可以由 `java.lang.Package`、`java.lang.Class` 实现, 也可以间接地由 `Method`、`Constructor` 以及 `java.lang.reflect` 的 `Field` 类所实现。在 `method` 参数里的注释可以使用 `Method` 或 `Constructor` 类的 `getParameterAnnotations()` `method` 来查询。

以下的程序代码使用了 `AnnotatedElement` 的 `isAnnotationPresent()` `method`, 通过检查 `@Unstable` 注释来判定 `method` 是否稳定。它假设该 `Unstable` 注释类型 (我们将在本章的稍后定义) 具有运行时 `retention`。请注意, 此程序代码使用了类直接量来指定被检查的类以及用来做检查的注释:

```
import java.lang.reflect.*;
Class c = WhizzBangClass.class;
Method m = c.getMethod("whizzy", int.class, int.class);
boolean unstable = m.isAnnotationPresent(Unstable.class);
```

对标志注释来说, `isAnnotationPresent()` 是有用的。一般来说, 当它与拥有成员的注释一起运行时, 我们会想要知道这些成员的值。针对这点, 我们可以使用 `getAnnotation()` `method`。同时, 在这里我们也看到了 Java 注释系统的优点: 如果被指定的注释存在的话, 通过此 `method` 返回的对象实现该注释类型的接口, 而且你也可以通过调用定义该成员的注释类型 `method` 来查询任何成员的值。例如, 考虑本章稍早



所出现的 @Reviews 注释。如果注释类型被声明为 runtime retention，那么你可以用以下的方式来查询：

```
AnnotatedElement target = WhizzBangClass.class; // 此类型是用来做查询的
// 要求 @Reviews 注释是一个实现 Reviews 的对象
Reviews annotation = target.getAnnotation(Reviews.class);
// Reviews 只有一个单一的名为“value”的成员，它是由 reviews 组成的数组
Review[] reviews = annotation.value();
// 循环处理 reviews
for(Review r : reviews) {
    Review.Grade grade = r.grade();
    String reviewer = r.reviewer();
    String comment = r.comment();
    System.out.printf("%s assigned a grade of %s and comment '%s'\n",
                      reviewer, grade, comment);
}
```

请注意，这些反射 method 可以为你正确地解决默认注释值的问题。如果注释没有为成员提供默认值，则该默认值可以在注释类型里被查询到。

## 定义注释类型

注释类型是一个接口，但并不是正常的接口。注释类型与正常的接口有以下几个不同的地方：

- 注释类型是使用关键字 @interface 来定义，而不是 interface。@interface 的声明会隐含地扩展 java.lang.annotation.Annotation 接口，且在它的定义里不能有它自己的 extends 子句。
- 注释类型的 method 必须被声明为没有任何的自变量而且不可以抛出异常。这些 method 定义了注释成员：method 的名称变成了成员的名称，同时 method 的返回类型变成了成员的类型。
- 注释 method 的返回值可以是基本类型、String、Class、枚举类型、另一个注释类型或是这些类型的一维数组。
- 注释类型的任何 method 后面可以紧跟着关键字 default 以及与该 method 返回类型兼容的值。这个新的语法指定了与该 method 相应的注释成员的默认值。在写注释的时候，默认值的语法与用来指定成员值的语法是一样的。null 不是一个合法的默认值。
- 注释类型与它们的 method 不可以有类型参数注释类型（parameters-annotation type），而且它的成员不可以是泛型。在注释类型里唯一可以合法使用泛型的是返回类型为 Class 的 method。这些 method 可以使用有限制的通配符来指明返回类型的约束。

在其他方式中，注释类型使用了@interface来声明就像一般的接口一样。它可以包含常量的定义以及静态成员类型的定义，例如枚举类型的定义。注释类型也可以像正常的接口一样被实现与扩展（然而，实现或扩展该类与接口的结果并不是它们自己的注释类型：注释类型只可以被创建于@interface的声明里）。

现在我们在自己的范例里定义注释类型。这些范例说明了注释类型声明的语法并显示了@interface与interface之间的多处不同点。我们从简单的标志注释类型Unstable开始，因为在本章稍早的一个reflection范例里使用到这个类型，它的定义包含了一个meta-annotation，会将它设定为runtime retention并让它可以使用reflection API。下面会对meta-annotation做说明。

```
package com.davidflanagan.annotations;
import java.lang.annotation.*;

/**
 * 指定被注释的元素是不稳定的而且它的API会改变
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface Unstable {}
```

接下来的注释类型定义了一个单一的成员。通过命名成员value，我们就能对每个使用注释的程序使用更简短的语句。

```
/**
 * 指明程序元素的作者
 */
public @interface Author {
    /** 返回作者的名字 */
    String value();
}
```

再下来的一个范例就有点复杂了。Reviews注释类型有一个单一的成员，但成员类型有点复杂：它是Review注释组成的数组。Review注释类型有三个成员，其中一个具有被定义成Review类型本身的成员的枚举类型，另一个具有默认值。因为Reviews注释类型被使用在reflection范例里，所以我们也用meta-annotation来给予它runtime retention：

```
import java.lang.annotation.*;

/**
 * 此类型的注释为注释元素指定了一个或多个程序代码review的结果
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface Reviews {
    Review[] value();
}
```

```
/**
 * 此类型的注释表示该注释元素的单一程序代码 review。
 * 每一个 review 必须指明 reviewer 的名称并为程序代码指定级别。
 * review 也可以包含一个注释 (comment) 字符串
 */
public @interface Review {
    // 嵌套的枚举类型
    public static enum Grade { EXCELLENT, SATISFACTORY, UNSATISFACTORY };

    // 这些 method 定义了注释成员
    Grade grade(); // Grade 类型里一个名为 "grade" 的成员
    String reviewer();
    String comment() default ""; // 请注意, 默认值在这里
}
```

最后, 假设我们想要为 method 做注释以列出它们可能抛出的未校验的异常 (不是错误)。我们的注释类型将会有有一个数组类型的单一成员, 该数组中的每一个元素将会是异常的 Class。为了只使用未校验的异常 (unchecked exception) 这个必要条件, 我们在 Class 里使用了一个受限制的通配符:

```
public @interface UncheckedExceptions {
    Class<? extends RuntimeException>[] value();
}
```

## Meta-Annotation

注释类型本身可以被注释。Java 5.0 定义了四个标准的 meta-annotation 类型, 它们提供了关于其他注释类型的使用与意义的信息。这些类型以及它们所支持的类都在 `java.lang.annotation` 包里。

## Target

Target meta-annotation 类型为注释类型指定了“目标 (target)”, 也就是它指定了哪一个程序元素可以有该类型的注释。如果一个注释类型没有 Target meta-annotation, 那么它就可以与稍早所介绍的任何程序成员一起使用。然而, 有些注释类型只有在当它被用于某个程序元素里时才具有意义。Override 就是一个例子: 当它被使用于一个 method 时才具有意义。当 @Target meta-annotation 被用于 Override 类型的声明时会让它更明确, 而且当它出现在不适当的地方时, 编译器会拒绝 @Override。

Target meta-annotation 类型有一个名为 value 的单一成员。此成员的类型为 `java.lang.annotation.ElementType[]`。ElementType 是一个枚举类型, 它的枚举值代表了可以被注释的程序元素。



## Retention

稍早我们在本章已经讨论过注释保留 (annotation retention) 了。它明确地指出注释是被编译器抛弃或被保留于类文件中, 以及如果它被保留在类文件里, 则当该类文件被加载时, 它是否会被 JVM 所读取。在默认的状态下, 注释是被存储于类文件里, 但不能在运行时取得这些注释。这三个可选的保留值为 `source`、`class` 与 `runtime`, 说明于枚举类型 `java.lang.annotation.RetentionPolicy` 中。

`Retention` meta-annotation 类型有一个名为 `value` 的单一成员, 它的类型为 `RetentionPolicy`。

## Documented

`Documented` 是个 meta-annotation 类型, 用来说明其他类型的注释。它应该被视为该注释的程序元素的公共 API 的一部分, 因此它可以被像 *javadoc* 这样的工具做成说明文件。`Documented` 是个标记注释, 它没有成员。

## Inherited

`@Inherited` meta-annotation 是个标志注释, 它说明了注释类型是会被继承的。也就是说, 如果注释类型 `@Inherited` 被用来帮类做注释时, 那么此注释一样适用于该类的子类。

请注意, `@Inherited` 注释类型只可以被注释类的子类所继承。类并不会从它们所实现的接口里继承注释, 而且 `method` 也不会从它们所覆盖的 `method` 里继承注释。

如果 `@Inherited` 注释类型也是一个有注释的 `@Retention(RetentionPolicy.RUNTIME)` 时, 则 `Reflection` API 会去执行继承。如果你使用了 `java.lang.reflect` 来为一个 `@Inherited` 类型的注释查询类时, 则 `reflection` 程序代码会去检查那个被指明的类以及它的每一个父类, 直到那个被指明类型的注释被找到或已经到达类层次结构的最顶层。



第二、三、四章说明了 Java 程序设计语言。本章转移到说明 Java 平台——为每个 Java 程序提供了大量可用的预定义类，无论它在运行时所在的底层主机系统是哪一种。Java 平台的类被组织成相关的组，也就是包（package）。本章从 Java 平台的包的概述开始，接着会以简短范例的形式来示范这些包中最有用的类。大部分的范例都只是程序代码片段，而不是可以编译并运行的完整程序。关于完全充实、真实世界的范例，请参阅《Java Examples in a Nutshell》（O'Reilly）。那本书大幅扩充了本章的内容，可作为本书的参考书籍。

## Java 平台概述

表 5-1 归纳了本书中涉及的 Java 平台的重要包。

表 5-1: Java 平台的重要包

包	说明
<code>java.io</code>	用于输入和输出的类和接口。虽然此包中有一些类是用来直接处理文件，但大部分是用来处理字节或字节流
<code>java.lang</code>	Java 语言的核心类，例如 <code>String</code> 、 <code>Math</code> 、 <code>SystemThread</code> 以及 <code>Exception</code>
<code>java.lang.annotation</code>	annotation 类型和其他用于 Java 5.0 annotation 特性的支持类型（请参阅第四章）
<code>java.lang.instrument</code>	针对 Java 虚拟机安装（instrumentation）代理程序的支持类，代理程序可修改 JVM 运行中程序的字节码。这是 Java 5.0 中新引入的

表 5-1: Java 平台的重要包 (续)

包	说明
java.lang.management	这是用于监控与管理运行中的 Java 虚拟机的结构, 是 Java 5.0 中新引入的
java.lang.ref	这些是定义对对象弱引用 (weak reference) 的类。弱引用并无法避免被引用对象被内存回收
java.lang.reflect	能让 Java 程序通过查看类的构造函数、method 和字段来回顾其自身的类和接口
java.math	这是个小型的包, 包含可用于任意精确度的整数和浮点运算的类
java.net	用于实现与其他系统建立网络连接的类和接口
java.nio	针对 New I/O API 的缓冲类。这是 Java 1.4 中新增的
java.nio.channels	针对高性能、非阻隔性 I/O 的 channel 和 selector 接口与类
java.nio.charset	用于在 Unicode 字符串和字节之间转换的字符组编码器与译码器
java.security	用于访问控制与认证的类与接口。此包及其子包支持加密消息摘要与数字签名
java.text	用于处理国际化应用程序中文本的类和接口
java.util	各种公用程序类, 其中包括用于处理大量对象的强大收集框架
java.util.concurrent	用于并行程序设计的线程共享区和其他的公用程序类, 子包支持 atomic 变量和锁定。这是 Java 5.0 新增的
java.util.jar	用于读取与写入 JAR 文件的类
java.util.logging	具有灵活性的登录工具。这是 Java 1.4 中新增的
java.util.prefs	用来读取与写入用户与系统的参数选择的 API。这是 Java 1.4 中新增的
java.util.regex	使用了正则表达式表达式的文本模式匹配。这是 Java 1.4 中新增的
java.util.zip	用于读取与写入 ZIP 文件的类
javax.crypto	针对数据加密与解密的类与接口
javax.net	定义用于建立 socket 和 server socket 的 factory class, 以供建立有别于默认的 socket 类型
javax.net.ssl	使用 Secure Sockets Layer (SSL) 来加密网络通信的类
javax.security.auth	用于认证与授权的 JAAS API 的顶层包, 各种子包掌握了大部分的实现类。这是 Java 1.4 中新增的



表 5-1: Java 平台的重要包 (续)

包	说明
<code>javax.xml.parsers</code>	用于解析使用了可插拔 DOM 或 SAX 解析器的 XML 文件的高级 API
<code>javax.xml.transform</code>	这是个高级 API, 用于转换使用了可插拔 XSLT 转换引擎和在流、DOM 树、SAX 事件之间转换 XML 文件, 子包对 DOM、SAX 和流转换提供了支持。这是 Java 1.4 中新增的

表 5-1 并没有列出 Java 平台中的所有包, 只列出本书中说明的最重要的包。Java 也定义了许多针对制图和图形用户界面设计以及用于分布式、企业或计算的包。制图和 GUI 包是 `java.awt` 和 `javax.swing` 以及它们的许多子包。这些包的说明文档在《Java Foundation Classes in a Nutshell》和《Java Swing》中有介绍, 这两本书都是由 O'Reilly 出版。Java 的企业包包括了 `java.rmi`、`java.sql`、`javax.jndi`、`org.omg.CORBA`、`org.omg.cosNaming`、以及它们的所有子包。这些包以及对 Java 平台的好几种标准扩展在《Java Enterprise in a Nutshell》(O'Reilly) 中有介绍。

## 文本

大部分的程序都会以多种方式来操作文本, 而 Java 平台针对表示、格式化以及扫描文本定义了一些重要的类和接口。以下的章节提供了一个概述。

## String 类

由文本组成的字符串是基本而常被使用的数据类型。但是在 Java 中, 字符串并不像 `char`、`int` 和 `float` 这样的基本类型。字符串类型是由 `java.lang.String` 类来定义的, 它定义了许多用于操作字符串的有用 `method`。`String` 对象是永远不变的: 一旦 `String` 对象被创建, 就无法修改它所表示的文本字符串。因此, 在字符串上操作的每个 `method` 通常会返回一个保存已修改字符串的新 `String` 对象。

下面的程序代码展示了一些你可以在字符串上执行的基本操作:

```
// 创建字符串
String s = "Now";                // String 对象具有特殊的直接量语法
String t = s + " is the time.";  // 使用 + 运算符来连接字符串
String t1 = s + " " + 23.4;      // 运算符 + 会把其他类型的值转换为字符串类型
t1 = String.valueOf('c');        // 取得对应每个字符的字符串
t1 = String.valueOf(42);         // 取得字符串类型的整数或任一值
t1 = object.toString();          // 使用 toString() 将对象转为字符串
```

```

// 字符串长度
int len = t.length();           // 字符串中的字符个数为 16

// 字符串的子字符串
String sub = t.substring(4);    // 返回从位置 4 开始到结尾的字符串的子串 "is the time."
sub = t.substring(4, 6);        // 返回位置 4 和 5 的字符 "is"
sub = t.substring(0, 3);        // 返回位置 0 到 2 的字符 "Now"
sub = t.substring(x, y);        // 返回介于位置 x 和 y-1 之间的字符串的子串
int numchars = sub.length();    // 子字符串的长度一定是 (y-x)

// 从字符串中提取字符
char c = t.charAt(2);           // 取得字符串 t 的第 3 个字符 w
char[] ca = t.toCharArray();    // 将字符串转换为字符数组
t.getChars(0, 3, ca, 1);        // 将字符串 t 的前三个字符放入 ca[1]-ca[3]

// 大小写转换
String caps = t.toUpperCase();  // 转换为大写
String lower = t.toLowerCase(); // 转换为小写

// 字符串比较
boolean b1 = t.equals("hello"); // 若两字符串不相等, 返回 false
boolean b2 = t.equalsIgnoreCase(caps); // 忽略大小写的比较, 若相等则返回 true
boolean b3 = t.startsWith("Now"); // 返回 true
boolean b4 = t.endsWith("time."); // 返回 true
int r1 = s.compareTo("Pow");     // 返回值 < 0, 则 s 出现在 "Pow" 之前
int r2 = s.compareTo("Now");     // 返回 0, 则两字符串相等
int r3 = s.compareTo("Mow");     // 返回值 > 0, 则 s 在 "Mow" 之后
r1 = s.compareToIgnoreCase("pow"); // 返回值 < 0 (适用于 Java 1.2 及之后的版本)

// 查找字符与子字符串
int pos = t.indexOf('i');        // 第一个 "i" 的位置: 4
pos = t.indexOf('i', pos+1);     // 下一个 "i" 的位置: 12
pos = t.indexOf('i', pos+1);     // 字符串中不再有 "i", 返回 -1
pos = t.lastIndexOf('i');       // 字符串中最后一个 "i" 的位置: 12
pos = t.lastIndexOf('i', pos-1); // 从位置 11 的字符开始由后往前查找

pos = t.indexOf("is");           // 查找子字符串: 返回 4
pos = t.indexOf("is", pos+1);    // 只出现一次: 返回 -1
pos = t.lastIndexOf("the ");    // 由后往前查找字符串
String noun = t.substring(pos+4); // 取出 "the" 之后的单词

// 以一个字符取代另一个字符
String exclam = t.replace('.', '!'); // 只能处理字符, 不能处理子字符串

// 清除字符串开头和结尾的空格
String noextraspaces = t.trim();

// 使用 intern() 取得字符串的唯一实例
String s1 = s.intern();          // 返回与 s 相等的字符串 s1
String s2 = "Now";              // 字符串直接量会自动被保留
boolean equals = (s1 == s2);     // 现在可以用 == 来比较是否相等

```

## Character 类

就如你所知道的, 每个字符在 Java 中是以基本的 char 类型来表示的。此外, Java 平台还定义了 Character 类, 它包含了用来检查字符类型和用于转换字符大小写的有用的类 method。例如:

```
char[] text; // 由字符组成的数组, 会在其他地方被初始化
int p = 0;    // 当前我们在字符数组中的位置
// 略过前面的空白
while((p < text.length) && Character.isWhitespace(text[p])) p++;
// 将文本的第一个字改为大写
while((p < text.length) && Character.isLetter(text[p])) {
    text[p] = Character.toUpperCase(text[p]);
    p++;
}
```

## StringBuffer 类

由于 String 对象是永远不变的, 所以你不可以操作已实例化的 String 字符。如果你必须这么做, 可以用 java.lang.StringBuffer 或 java.lang.StringBuilder 来代替。除了 StringBuffer 具有同步化 method 之外, 这两个类是完全一样的。StringBuilder 是在 Java 5.0 中引入的, 你应该用它来取代 StringBuffer, 除非实际上有多个线程在操作 StringBuffer。以下的程序代码示范了 StringBuffer API, 不过改变为使用 StringBuilder 也很容易:

```
// 根据字符串创建字符串缓冲区
StringBuffer b = new StringBuffer("Mow");

// 取得并修改 StringBuffer 的单独字符
char c = b.charAt(0);           // 返回 "M", 就像 String.charAt()
b.setCharAt(0, 'N');           // b 包含子串 "Now", 但是不能对字符串做这样的操作!

// 添加内容至 StringBuffer
b.append(' ');                  // 添加一个字符
b.append("is the time.");       // 添加一个字符串
b.append(23);                   // 添加一个整数或任何其他类型的值

// 将 String 或其他类型的值插入 StringBuffer 中
b.insert(6, "n't");             // b 现在包含子串 "Now isn't the time.23"

// 用一个字符串来代替一组字符 (适用于 Java 1.2 及之后的版本)
b.replace(4, 9, "is");          // 子串 b 恢复为 "Now is the time.23"

// Delete characters
b.delete(16, 18);               // 删除一组字符 "Now is the time"
b.deleteCharAt(2);              // 删除第 2 个字符, 字符串 b 变为 "No is the time"
b.setLength(5);                // 设定字符串的长度以截短字符串, 返回值为 "No is"
```



```
// 其他有用的操作
b.reverse();           // 逆序排列字符 "si oN"
String s = b.toString(); // 转换为不可变动的字符串
s = b.substring(1,2);   // 取得子字符串 "i"
b.setLength(0);        // 清空缓冲区, 现在它已可再被使用
```

## CharSequence 接口

在 Java 1.4 中, `String` 和 `StringBuffer` 类实现了 `java.lang.CharSequence` 接口, 那是个标准接口, 用于从具有可读性的一连串字符中查询长度、提取字符及后续项目。此接口也是由 `java.nio.CharBuffer` 接口实现的, 那是 Java 1.4 中引入的 New I/O API 的一部分。`CharSequence` 提供了在由字符组成的字符串上执行简单操作的方式, 而不管那些东西的底层实现。例如:

```
/**
 * 返回特定 CharSequence 的前缀, 由序列的第一个字符
 * 开始, 扩展至 (并包括) 序列中第一个出现的字符 c。
 * 如果 c 没有出现就返回 null。s 可以是 String、
 * StringBuffer 或 java.nio.CharBuffer。
 */
public static CharSequence prefix(CharSequence s, char c) {
    int numChars = s.length();           // 此序列有多长?
    for(int i = 0; i < numChars; i++) { // 逐个处理序列中的字符
        if (s.charAt(i) == c)           // 如果找到 c,
            return s.subSequence(0,i+1); // 就返回前缀子序列
    }
    return null;                         // 否则, 就返回 null
}
```

## Appendable 接口

`Appendable` 是 Java 5.0 中的接口, 它表示对象可以添加一个字符或 `CharSequence`。实现的类包括了 `StringBuffer`、`StringBuilder`、`java.nio.CharBuffer`、`java.io.PrintStream` 和 `java.io.Writer` 及其字符输出流子类, 其中包括 `PrintWriter`。因此, `Appendable` 接口表示文本缓冲区类和文本输出流类的可添加性。就如我们在下面看到的, `Formatter` 对象可以把它的输出送到任一个 `Appendable` 对象。

## String 连接

`+` 运算符可以连接两个 `String` 对象或一个 `String` 对象与一些其他类型的值并产生新的 `String` 对象。要知道, 每当执行字符串连接而且结果被存储于变量中或被传送给 `method` 时, 新的 `String` 对象就会被创建。在某些情况中, 这可能会没有效率而且造成性能不佳。在循环中进行字符串连接时更是要多加小心。例如, 以下的程序代码就没有效率:

```
// 无效率，别这么做
public String join(List<String> words) {
    String sentence = "";
    // 每一轮都会创建新的 String 对象并舍弃旧的
    for(String word: words) sentence += word;
    return sentence;
}
```

当你发现自己写出了像是这样的程序代码时，就应使用 `StringBuffer` 或 `StringBuilder` 并使用 `append()` method:

```
// 这是正确的方法
public String join(List<String> words) {
    StringBuilder sentence = new StringBuilder();
    for(String word: words) sentence.append(word);
    return sentence.toString();
}
```

但是，对字符串连接有所偏见是不必要的。请记住，字符串直接量是由编译器连接的，而不是由 Java 解释器连接的。此外，当字符串表达式包含多个字符串连接时，使用 `StringBuilder` (或 Java 5.0 之前所提供的 `StringBuffer`) 在编译上较有效率，并且只会创建出一个新的 `String` 对象。

## 字符串比较

由于字符串是对象而不是基本的值，所以一般说来，不能用 `==` 运算符来比较它们。`==` 会比较引用值，并且可以判定两个已引用的表达式是否为相同的字符串。它无法判定两个独立的字符串是否包含相同的文字。如果要这么做，可以用 `equals()` method。在 Java 5.0 中，你可以用 `contentEquals()` method 来将一个字符串的内容与任何其他 `CharSequence` 比较。

同样地，`<` 和 `>` 关系运算符也不可用于字符串。如果要比较字符串的顺序，可使用 `compareTo()` method，那是由 `Comparable<String>` 接口所定义的，并在上面的范例程序代码中说明过。如果在比较字符串时不考虑字母的大小写，可以用 `compareToIgnoreCase()`。

请注意，`StringBuffer` 和 `StringBuilder` 没有实现 `Comparable`，而且没有覆写默认版本的 `equals()` 和 `hashCode()`，那是它们从 `Object` 继承来的。这表示要对两个 `StringBuffer` 或 `StringBuilder` 包含的文字作相等性或顺序性比较是不可能的。

`String` 类中有一个很重要但很少被了解的 method，那就是 `intern()`。当字符串 `s` 被传递给这个 method 时，它会返回保证与 `s` 具有相同内容的字符串。但重要的是，对于任

意一个给定的字符串的内容，它一定会返回对相同 `String` 对象的引用。也就是说，如果 `s` 和 `t` 是两个如 `s.equals(t)` 这样的 `String` 对象，则：

```
s.intern() == t.intern()
```

这代表 `intern()` method 使用 `==` 提供了一个进行快速比较字符串的方法。重要的是，字符串直接量一定会由 Java VM 隐含地保留，所以如果你计划比较字符串 `s` 和一些字符串直接量，你或许会想要先保留 `s`，然后再用 `==` 作比较。

字符串类的 `compareTo()` 和 `equals()` method 能让你比较字符串。`compareTo()` 是由 Unicode 编码的字符顺序作为比较基础，而 `equal()` 把字符串的相等定义为严格的逐字符相等。但是在使用上，这些未必都是适当的 method。在某些语言中，Unicode 标准所定的字符排序，与以字母顺序排序字符串时的字排序不同。例如，在西班牙文中，字母“ch”会被认为是出现在“c”之后、“d”之前的独立字母。在国际化应用程序中比较人类可阅读的字符串时，你应该用 `java.text.Collator` 类来代替：

```
import java.text.*;

// 比较两个字符串，结果取决于运行程序的地方
// Collator.compareTo() 的返回值具有与 String.compareTo() 相同的意义
Collator c = Collator.getInstance(); // 取得当前本地的 Collator
int result = c.compare("chica", "coche"); // 用它来比较两个字符串
```

## 增补字符

Java 5.0 首次采用了 Unicode 4.0 标准，它定义了落在 `char` 类型的 16 位范围之外的码点 (codepoint)。当处理这些“增补字符 (supplementary characters)”时 (这些主要是中文)，你必须使用 `int` 值来代表各个字符。在 `String` 对象或是任何其他将文字表示成一连串 `char` 值的类型中，这些增补字符会被表示成一连串的双 `char` 值，也就是所谓的 *surrogate pair*。

虽然本书的英文版读者不太可能遇到增补字符，但如果你要处理的程序可能会被本地化以用于中国或其他使用中文的国家时，就应该要有所了解。为了帮助你处理增补字符，`Character`、`String`、`StringBuffer` 以及 `StringBuilder` 类已用新 method 来加以扩充，那些 method 是在 `int` 码点上操作，而不是在 `char` 值上。以下程序代码说明了其中的一些 method。你可以在在线的 javadoc 说明文件中阅读其他类似的 method 的相关信息。

```
int codepoint = 0x10001; // 此码点无法被放入一个 char 中
// 取得此码点的 char 的 UTF-16 代理对
char[] surrogatePair = Character.toChars(codepoint);
// 将 char 转换为字符串
String s = new String(surrogatePair);
```



```
// 列出以字符和码点计算的字符串长度
System.out.println(s.length());
System.out.println(s.codePointCount(0, s.length()-1));

// 先列出第一个字符的编码, 接着列出第一个码点的编码
System.out.println(Integer.toHexString(s.charAt(0)));
System.out.println(Integer.toHexString(s.codePointAt(0)));

// 这是安全处理可能包含增补字符的字符串方式
String tricky = s + "Testing" + s + "!";
int i = 0, n = tricky.length();
while(i < n) {
    // 取得当前位置的码点
    int cp = tricky.codePointAt(i);
    if (cp < '\uffff') System.out.println((char) cp);
    else System.out.println("\u" + Integer.toHexString(cp));

    // 以一个码点 (一或两个字符) 来递增字符串索引
    i = tricky.offsetByCodePoints(i, 1);
}
```

## 用 printf() 和 format() 来格式化文字

处理文字输出时的常见工作, 就是把各种类型的值结合成人能看懂的独立文字块。完成此工作的方法之一, 就是依赖Java字符串连接运算符的字符串转换能力。那会产生像这样的程序代码:

```
System.out.println(username + " logged in after " + numattempts +
    " attempts. Last login at: " + lastLoginDate);
```

Java 5.0 引进了C程序员熟悉的替代方法: printf() method。“printf”是“print formatted”的缩写, 它把打印与格式化功能结合在一个调用中。printf() method 已被加到Java 5.0 中的PrintWriter和PrintStream输出流类。它是个预期会有一个或多个自变量的varargs method。第一个自变量是“格式字符串”, 它指定了要被打印的文字, 通常包含了一个或多个“格式限定符 (format specifier)”, 那是以字符%开始的序列。printf()的其余自变量要被转换为字符串的值, 并且会替换格式字符串中的格式限定符。格式限定符限制了其余自变量的类型, 并且精确地指定了它们被转换为字符串的方式。上面显示的字符串连接在Java 5.0 中可以被改写如下:

```
System.out.printf("%s logged in after %d attempts. Last login at: %tc%n",
    username, numattempts, lastLoginDate);
```

格式限定符%s只是单纯替换一个字符串;%d预期相对应的自变量会是整数并加以显示;%tc预期相对应的值是Date、Calendar或毫秒数, 并把那个值转换为完整日期和时间的文字表达式;%n不会执行转换, 它只是输出平台特有的行结束字符, 就如println() method 所做的。

printf()所执行的转换都会被恰当地本地化。例如,时间和日期会以符合当地习惯的标点符号来显示。而且,如果你要求数值加上千分位符号显示,也会得到各地用的不同的标点符号(例如,在英国是逗号,在法国是句号)。

除了基本的printf() method,PrintWriter和PrintStream也定义了名为format()的同义method:它采用完全相同的自变量,行为方式也完全相同。在Java 5.0中,String类也有format() method。除了把已格式化的字符串输出到串流外,此静态String.format() method的行为就像PrintWriter.format()一样,它只是返回:

```
// 使用两位小数和千分位符号来格式化一个字符串,将double
// 值转换为文字
double balance = getBalance();
String msg = String.format("Account balance: $%,.2f", balance);
```

java.util.Formatter类是个在printf()和format() method后面的多用途格式化类。它可以将文字格式化为任何Appendable对象或具名文件。以下程序代码使用了Formatter对象来写文件:

```
public static void writeFile(String filename, String[] lines)
    throws IOException
{
    Formatter out = new Formatter(filename); // format to a named file
    for(int i = 0; i < lines.length; i++) {
        // 写一行到文件
        out.format("%d: %s\n", i, lines[i]);
        // 检查异常
        IOException e = out.ioException();
        if (e != null) throw e;
    }
    out.close();
}
```

当你将对象与字符串连接时,调用对象的toString() method可以将它转换为字符串。这也是Formatter类的默认动作。如果要对类的格式化做更精确的控制,除了实现toString()之外,还可以实现java.util.Formatter接口。

在我们说明处理数值、日期以及时间的API时,会看到一些以printf()来进行格式化的额外范例。关于可使用的格式限定符和选项的完整列表,请参阅java.util.Formatter。

## Logging

基于终端机的单纯程序可以用System.out.println()或System.out.print()来传送输出结果和错误信息到控制台。无人看管且长时间运行的服务器程序需要不同的输出方案:它们在运行时所用的硬件上可能没有连接终端机来显示,如果真是如此,就不太

可能有任何人看着。这样的程序必须要有日志记录功能,输出信息要被传送到文件以供后续分析,或经由网络 socket 以供远程监控。Java 1.4 在 `java.util.logging` 包中提供了日志记录 API。

通常,应用程序开发者会使用与应用程序的类或包相结合的 `Logger` 对象来产生 7 种安全等级 (severity level) 的日志信息 (请参阅 `java.util.logging.Level`)。这些信息可以报告错误和警告,或提供有关应用程序生命周期中令人感兴趣的事件的信息。它们可以包含调试信息,甚至追踪程序中重要 `method` 的执行。

应用程序的系统管理员或终端用户有责任设定日志记录配置文件,其中设定了日志信息被导向何处 (控制台、文件、网络 socket 或这些的组合)、格式化的方式 (一般文本文件或 XML 文件) 以及它们被记录时的安全门槛 (severity threshold, 具有规定门槛以下的安全信息会在资源极少的情况下被抛弃,而且不应该明显影响应用程序的性能)。日志记录安全门槛等级可以被独立地设定,以便与不同类或包结合的 `Logger` 对象能被“调入”或“调出”。由于具有可调性,你应该放心地在你的程序中自由使用日志记录输出。在正常运行中,大部分的日志信息会被有效率地自动抛弃。但是,在程序开发期间或在诊断已配置的应用程序的问题时,日志信息被证明非常有价值。

对于大部分的应用程序来说,使用 `Logging API` 是相当简单的。每当有需要时,调用静态 `Logging.getLogger()` `method`、传送应用程序的类或包的名称作为 `logger` 名称,就能取得具名 `Logger` 对象。接着,使用多个 `Logger` 实例 `method` 中的一个来产生日志信息。最易于使用的 `method`,其名称是与安全等级对应的,例如 `severe()`、`warning()` 以及 `info()`。这里有一些范例程序代码:

```
import java.util.logging.*;

// 取得以当前包命名的 Logger 对象
Logger logger = Logger.getLogger("com.davidflanagan.servers.pop");
logger.info("Starting server.");           // 记录具有信息的信息
ServerSocket ss;                          // 做一些事
try { ss = new ServerSocket(110); }
catch (Exception e) {                    // 记录异常
    logger.log(Level.SEVERE, "Can't bind port 110", e); // 复杂的日志信息
    logger.warning("Exiting");             // 单纯的警告
    return;
}
logger.fine("got server socket"); // 详细 (低安全性) 的调试信息
```

## 使用正则表达式进行模式匹配

在 Java 1.4 及之后的版本中,你可以用正则表达式来执行文本模式匹配。正则表达式是由 `java.util.regex` 包的 `Pattern` 和 `Matcher` 类提供支持的,但 `String` 类定义了一



些方便的method,能让你在使用正则表达式时更为容易。正则表达式使用了相当复杂的文法来描述字符组成的模式。Java的实现使用了与Perl 5程序设计语言相同的regex语法。关于此语法的更进一步的细节,请查阅适合的Perl程序设计书籍。至于Perl风格的正则表达式的完整教学,请参阅《Mastering Regular Expressions》(O'Reilly)。

接受正则表达式自变量的最简单的 String method 就是 `matches()`。如果字符串与特定正则表达式所定义的模式相匹配，就返回 `true`：

```
// 此字符串是个正则表达式，它描述了典型句子
// 的模式。在 Perl 风格的正则表达式语法中，
// 这指定了以大写字母开始并以句号、问号、
// 或惊叹号结束的字符串
String pattern = "^[A-Z].*[\.\?\!]+$";
String s = "Java is fun!";
s.matches(pattern); // 此字符串与模式相同，所以会返回 true
```

`matches()` method 只有在整个字符串与指定的模式相匹配时才会返回 `true`。Perl 程序员应该注意，这与 Perl 的行为不同，在 Perl 中，相匹配只是代表字符串的一些部分与模式相匹配。如果要判定字符串或任何子字符串是否与模式相匹配，只要变动正则表达式以允许所需模式的前后有任意字符即可。在以下程序代码中，正则表达式字符 `.` 与任何数量的任意字符相匹配：

```
s.matches(".*\\bJava\\b.*"); // 如果s有任何地方包含单词“Java”就是true
                             // b指定了单词界限
```

如果你已熟悉 Perl 的正则表达式语法，就会知道它依赖自由使用反斜线来输出特定字符。在 Perl 中，正则表达式是语言原有的，它们的语法是语言本身的一部分。但是在 Java 中，正则表达式是用字符串来描述的，而且通常会使用字符串直接量来内嵌在程序中。Java 的字符串直接量的语法也使用了反斜线作为转义字符，所以如果要在正则表达式中包含一个反斜线，就必须使用两个反斜线。因此，在 Java 程序设计中，你常会在正则表达式中看到双反斜线。

除了匹配之外，正则表达式还可以用于搜索与取代操作。`replaceFirst()` 和 `replaceAll()` method 会搜索字符串中与给定模式相匹配的第一个子字符串或所有子字符串，并以指定的替换文字来取代那些字符串，再返回包含有替换文字的新字符串。例如，你可以用下面这段程序代码来确保单词“Java”在字符串 `s` 中正确地使用了大写：

```
s.replaceAll("(?i)\\bjava\\b", // 模式: 单词 "Java", 不区分大小写
            "Java");          // 替换字符串, 正确地使用大写
```

传递给 `replaceAll()` 和 `replaceFirst()` 的替换字符串不必是单纯的字符串直接量，它也可以包含对与模式中加上括号的子表达式相匹配的文本的引用。这些引用采用的形式是美元符号后面加上一些子表达式。例如，如果要搜索如 `JavaBean`、`JavaScript`、

JavaOS、和 Java VM 这样的词（不含 Java 或 Javanese）并用字母 J 取代 java，可以使用这样的程序代码：

```
s.replaceAll("\\bJava([A-Z]\\w+)", // 模式
            "JS1");           // J 后面的字符与括号中的匹配
                           // 子表达式:[A-Z]\\w+
```

另一个使用了正则表达式的 String method 是 split()，它会返回由字符串的子字符串所组成的数组，用来分隔的定界符与指定的模式相匹配。如果要取得字符串中由任意数量的空格、跳格符或换行分隔的字符串组成的数组，就这样做：

```
String sentence = "This is a\n\ttwo-line sentence";
String[] words = sentence.split("[ \\t\\n\\r]+");
```

选择性的第二个自变量指定了返回数组中项目的最大数量。当你只使用一次正则表达式时，matches()、replaceFirst()、replaceAll()以及 split() method 很适合。如果你想把正则表达式用于多重匹配，就应该显式地使用 java.util.regex 包中的 Pattern 和 Matcher 类。首先，以静态 Pattern.compile() method 创建 Pattern 对象表示你的正则表达式（另一个显式地使用 Pattern 类来代替 String 的 method 的原因就是 Pattern.compile() 能让你指定如 Pattern.CASE\_INSENSITIVE 这样的标记，来全部变动模式匹配的方式）。请注意，如果你输入无效的正则表达式字符串，则 compile() method 可以被抛出 PatternSyntaxException（此异常也会由各类 String method 抛出）。Pattern 类定义了与 String.split() method 类似的 split() method。但是，对于其他的匹配，你必须用 matcher() method 创建 Matcher 对象并指定所要匹配的文字：

```
import java.util.regex.*;

Pattern javaword = Pattern.compile("\\bJava(\\w*)",
Pattern.CASE_INSENSITIVE);
Matcher m = javaword.matcher(sentence);
boolean match = m.matches(); // 如果文字与模式完全匹配就返回 true
```

一旦你有了 Matcher 对象，就可以用各种方法来比较字符串和模式。较复杂的一种方式，就是找出所有与模式相匹配的子字符串：

```
String text = "Java is fun; JavaScript is funny.";
m.reset(text); // 开始与新字符串匹配
// 循环找出所有相匹配字符串，并列出每个相匹配部分的细节
while(m.find()) {
    System.out.println("Found '" + m.group(0) + "' at position " +
m.start(0));
    if (m.start(1) < m.end(1)) System.out.println("Suffix is " +
m.group(1));
}
```

Matcher 类在 Java 5.0 中已用好几种方式来加以强化。这其中最重要的就是将最近的匹配结果存储在 MatchResult 对象中的能力。前面的找出字符串中所有相匹配部分的算法，在 Java 5.0 中可以改写如下：

```
import java.util.regex.*;
import java.util.*;

public class FindAll {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile(args[0]);
        String text = args[1];

        List<MatchResult> results = findAll(pattern, text);
        for(MatchResult r : results) {
            System.out.printf("Found '%s' at (%d,%d)%n",
                              r.group(), r.start(), r.end());
        }
    }

    public static List<MatchResult> findAll(Pattern pattern, CharSequence text)
    {
        List<MatchResult> results = new ArrayList<MatchResult>();
        Matcher m = pattern.matcher(text);
        while(m.find()) results.add(m.toMatchResult());
        return results;
    }
}
```

## 文字标记化

java.util.Scanner 是个通用的文字标记化程序，在 Java 5.0 中被加入以与本章前面所说明的 java.util.Formatter 类互补。Scanner 充分运用了 Java 正则表达式，并且可以从字符串、文件、流或任何实现了 java.lang.Readable 接口的对象取得输入的文字。Readable 也是 Java 5.0 中新出现的，是 Appendable 接口的相反物。

Scanner 可以将输入的文字打散成为标记，这些标记以空白、任何你想要的分隔字符或正则表达式来进行分隔。它实现了 Iterator<String> 接口，这能对返回的标记进行简单的逐一处理。Scanner 也定义了各种方便的 method，使用依地区识别数值的解析方式，把标记解析为布尔值、整数或浮点值。skip() method 会略过与指定模式相匹配的输入文字，也有可以针对输入文字搜索与指定模式相匹配的文字的 method。

这里是使用 Scanner 将 String 打散为以空格分隔的单词的方式：

```
public static List<String> getTokens(String line) {
    List<String> result = new ArrayList<String>();
    for(Scanner s = Scanner.create(line); s.hasNext(); )
        result.add(s.next());
}
```



```
    return result;
}
```

这里是使用 Scanner 将文件打散为行的方式：

```
public static void printLines(File f) throws IOException {
    Scanner s = Scanner.create(f);
    // 使用 regex 将换行字符指定为标记分隔符
    s.useDelimiter("\\r\\n|\\n|\\r");
    while(s.hasNext()) System.out.println(s.next());
}
```

以下 method 使用 Scanner 来解析具有  $x + y = z$  形式的输入，它示范了 Scanner 扫描数值的能力。请注意，Scanner 不只是解析 Java 风格的整数直接量：它支持千分位符号并且是以依地区识别的方式来进行。例如，它会针对美国的用户解析整数 1,234，针对法国的用户解析 1.234。这段程序代码也说明了 skip() method，并展示了 Scanner 可以从 InputStream 直接扫描文字。

```
public static boolean parseSum() {
    System.out.print("enter sum> "); // 提示用户输入
    System.out.flush();              // 确定提示可立刻被看见

    try {
        // 读取并解析来自控制台的用户输入
        Scanner s = Scanner.create(System.in);
        s.useDelimiter("");          // 不要求标记间要有空格
        int x = s.nextInt();          // 解析整数
        s.skip("\\s*\\+\\s*");        // 略过非必要的空格和直接量 +
        int y = s.nextInt();          // 解析另一个整数
        s.skip("\\s*=\\s*");          // 略过非必要的空格和直接量 =
        int z = s.nextInt();          // 解析第三个整数

        return x + y == z;
    }
    catch(InputMismatchException e) { // 无匹配的模式
        throw new IllegalArgumentException("syntax error");
    }
    catch(NoSuchElementException e) { // 不再有输入可用
        throw new IllegalArgumentException("syntax error");
    }
}
```

## StringTokenizer

还有其他一些 Java 类会分隔字符串和字符。一个值得注意的类是 java.util.StringTokenizer，你可以用它将文字字符串打散为组成其的单词：

```
String s = "Now is the time";
java.util.StringTokenizer st = new java.util.StringTokenizer(s);
```

```
while(st.hasMoreTokens()) {  
    System.out.println(st.nextToken());  
}
```

你甚至可以使用这个类来标记用空格以外的字符所分隔的单词：

```
String s = "a:b:c:d";  
java.util.StringTokenizer st = new java.util.StringTokenizer(s, ":");
```

java.io.StreamTokenizer是另一个标记类，它具有比StringTokenizer更复杂的API并且具有更强大的性能。

## 数值与数学运算

Java提供了byte、short、int、long、float和double这些基本类型来表示数值。java.lang包包含了对应的Byte、Short、Integer、Long、Float以及Double类，这些都是Number的子类。这些类在作为其基本类型的对象封装程序时很有用，而且它们也定义了一些有用的常量：

```
// 整数范围的常量：Integer、Long以及Character也定义了这些  
Byte.MIN_VALUE      // 最小（最负）的byte值  
Byte.MAX_VALUE      // 最大的byte值  
Short.MIN_VALUE     // 最小的short值  
Short.MAX_VALUE     // 最大的short值  
  
// 浮点数范围的常量：Double也定义了这些  
Float.MIN_VALUE     // 最小（最接近零）的正浮点值  
Float.MAX_VALUE     // 最大的正浮点值  
  
// 其他有用的常量  
Math.PI             // 3.14159265358979323846  
Math.E              // 2.7182818284590452354
```

## 数学函数

Math类定义了一些method，提供三角、对数、指数、四舍五入运算等。此类主要是在浮点值运算时很有用。针对三角函数，角度是以弧度制来表示。对数和指数函数是以 $e$ 为底数，不是以10为底数。这有一些范例：

```
double d = Math.toRadians(27);    // 将角度值 27 度转为弧度值  
d = Math.cos(d);                  // 取得余弦值  
d = Math.sqrt(d);                 // 取得平方根  
d = Math.log(d);                  // 取得自然对数  
d = Math.exp(d);                  // 计算它的反函数：取 e 的 d 次方  
d = Math.pow(10, d);              // 取 10 的 d 次方  
d = Math.atan(d);                 // 计算反正切值  
d = Math.toDegrees(d);            // 再把弧度值转换为角度值
```

```
double up = Math.ceil(d);           // 无条件进位
double down = Math.floor(d);        // 无条件舍去
long nearest = Math.round(d);       // 四舍五入至最近的整数
```

在 Java 5.0 中, Math 类中增加了几个新的函数, 其中包括以下函数:

```
double d = 27;
d = Math.cbrt(d);           // 立方根
d = Math.log10(d);          // 以 10 为底的对数
d = Math.sinh(d);           // 双曲线正弦, 还有 cosh() 和 tanh()
d = Math.hypot(3, 4);       // 直角三角形的斜边 (Hypotenuse)
```

## 随机数

Math 类也定义了用于产生虚拟随机数的基本 method, 但 java.util.Random 类较具灵活性。如果你需要随机性很强的虚拟随机数, 可以使用 java.security.SecureRandom 类:

```
// 简单的随机数
double r = Math.random();        // 返回: 0.0 <= d < 1.0

// 创建一个新的 Random 对象, 以当前时间作为参数
java.util.Random generator = new
java.util.Random(System.currentTimeMillis());
double d = generator.nextDouble(); // 0.0 <= d < 1.0
float f = generator.nextFloat();   // 0.0 <= f < 1.0
long l = generator.nextLong();     // 从整个 long 范围内选择
int i = generator.nextInt();        // 从整个 int 范围内选择
i = generator.nextInt(limit);       // 0 <= i < limit (适用于 Java 1.2 及之后的
                                    // 版本)
boolean b = generator.nextBoolean(); // true 或 false (适用于 Java 1.2 及之后的版本)
d = generator.nextGaussian();        // 平均值为 0.0, 标准方差为 1.0
byte[] randomBytes = new byte[128];
generator.nextBytes(randomBytes);    // 把随机 byte 填入数组

// 针对加密的随机数, 使用 SecureRandom 子类
java.security.SecureRandom generator2 = new java.security.SecureRandom();
// 让产生器产生它自己的 16 字节参数, 这会花很长的时间
generator2.setSeed(generator2.generateSeed(16)); // 额外的随机 16 字节参数
// 接着就像其他的 Random 对象一样使用 SecureRandom
generator2.nextBytes(randomBytes);      // 产生更多的随机字节
```

## 数值很大的数

java.math 包包含了 BigInteger 和 BigDecimal 类, 这些类能让你处理任意大小与任意精确度的整数和浮点数值。例如:

```
import java.math.*;

// 计算 1000 的阶乘
BigInteger total = BigInteger.valueOf(1);
```



```
for(int i = 2; i <= 1000; i++)
    total = total.multiply(BigInteger.valueOf(i));
System.out.println(total.toString());
```

在Java 1.4中, BigInteger有一个method来随机地产生很大的原始数值, 这在许多的密码应用中很有用:

```
BigInteger prime =
    BigInteger.probablePrime(1024,          // 1024 位长
                           generator2); // 随机数的来源, 来自于上文
```

BigDecimal类在Java 5.0中已被彻底修改, 在这版中更有用。它除了能代表非常大或非常精确的浮点数之外, 在财务计算中也很有用, 因为财务计算是用十进制的表示方式, 而不是二进制的表示方式。float和double值无法精确地将数值简单地表示为0.1, 这会造成四舍五入错误, 通常在表示货币值时是不能接受的。BigDecimal与其相关的MathContext和RoundingMode类型提供了解决方案。例如:

```
// 计算贷款所需付的月利率
public static BigDecimal monthlyPayment(int amount, // 贷款金额
                                         int years,  // 年限
                                         double apr) // 年利率 %
{
    // 将贷款金额转换为 BigDecimal
    BigDecimal principal = new BigDecimal(amount);

    // 将以年计算的贷款期限转换为月
    int payments=years*12;

    // 将利息从年利率转换为月利率
    BigDecimal interest = BigDecimal.valueOf(apr);
    interest = interest.divide(new BigDecimal(100)); // 成为小数
    interest = interest.divide(new BigDecimal(12));  // 每月

    // 月付款计算
    BigDecimal x = interest.add(BigDecimal.ONE).pow(payments);
    BigDecimal y = principal.multiply(interest).multiply(x);
    BigDecimal monthly = y.divide(x.subtract(BigDecimal.ONE),
                                  MathContext.DECIMAL64); // 请注意来龙去脉

    // 转换为两位小数
    monthly = monthly.setScale(2, RoundingMode.HALF_EVEN);

    return monthly;
}
```

## 在数字与字符串之间转换

在数字上进行操作的Java程序必须从某些地方取得输入值。通常, 这样的程序会读取文本形式的数字, 而且必须将它转换成数字形式。各种Number子类定义了很多有用的转换method:

```
String s = "-42";
byte b = Byte.parseByte(s);           // s 作为 byte
short sh = Short.parseShort(s);       // s 作为 short
int i = Integer.parseInt(s);           // s 作为 int
long l = Long.parseLong(s);            // s 作为 long
float f = Float.parseFloat(s);         // s 作为 float (适用于 Java 1.2 及之后的版本)
f = Float.valueOf(s).floatValue();     // s 作为 float (适用于 Java 1.2 及之后的版本)
double d = Double.parseDouble(s);      // s 作为 double (适用于 Java 1.2 及之后的版本)
d = Double.valueOf(s).doubleValue();    // s 作为 double (适用于 Java 1.2 之前的版本)

// 整数转换规则会处理使用其他进制的数字
byte b = Byte.parseByte("1011", 2);   // 二进制中的 1011 是十进制中的 11
short sh = Short.parseShort("ff", 16); // 十六进制中的 ff 是十进制中的 255

// valueOf() method 可以处理从 2 到 36 的任意基数
int i = Integer.valueOf("egg", 17).intValue(); // 17 进制!

// decode() method 能处理八进制、十进制或十六进制的数字, 具体使用的是哪种方式取决
// 于字符串的数字前缀
short sh = Short.decode("0377").byteValue(); // 以 0 开头代表八进制
int i = Integer.decode("0xff").shortValue(); // 以 0x 开头代表十六进制
long l = Long.decode("255").intValue();       // 其他的数字代表十进制

// Integer 类可以将数字转换为字符串
String decimal = Integer.toString(42);
String binary = Integer.toBinaryString(42);
String octal = Integer.toOctalString(42);
String hex = Integer.toHexString(42);
String base36 = Integer.toString(42, 36);
```

## 格式化数字

本章前面所说明的 Java 5.0 的 `printf()` 和 `format()` method 也能用于格式化数字。`%d` 格式限定符用于将整数格式化为十进制格式:

```
// 将 int、long 和 BigInteger 格式化为字符串 "1 10 100"
String s = String.format("%d %d %d", 1, 10L, BigInteger.TEN.pow(2));
// 增加千分位符号
s = String.format("%,d", Integer.MAX_VALUE); // "2,147,483,647"
// 将输出值向右对齐放在八个字符宽度的字段中
s = String.format("%8d", 123);                // "123"
// 以 0 填充左边以让数字总共有 5 个
s = String.format("%05d", 123);                // "00123"
```

浮点数可以用 `%f`、`%e` 或 `%g` 格式限定符来格式化, 这在是否使用指数符号时会有差别:

```
double x = 1.234E9; // 1.234 billion, 12.34 亿
// 返回 "1234000000.000000 1.234000e+09 1.234000e+09 1234.000000"
s = String.format("%f %e %g %g", x, x, x, x/1e6);
```

你会注意到以上的数字在小数点后都有六位。这样的默认值可以通过变更格式字符串中的精确度来更改:

```
// 显示具有两位小数的 BigDecimal
s = String.format("%.2f", new BigDecimal("1.234")); // "1.23"
```

其他的标记也可以应用至浮点转换。以下程序代码将数字栏格式化为宽为10个字符、向右对齐的格式。每个数字在小数点后都有两位，并且在有需要时加上千分位符号。负值会被格式化为加上括号，这是会计上常见的格式惯例。

```
// 由 4 个数字组成的栏。%n 是换行字符
s = String.format("%(,10.2f%n%(,10.2f%n%(,10.2f%n%(,10.2f%n",
    BigDecimal.TEN, // 10.00
    BigDecimal.TEN.movePointRight(3), // 10,000.00
    BigDecimal.TEN.movePointLeft(3), // 0.01
    BigDecimal.TEN.negate()); // (10.00)
```

在 Java 5.0 之前，可以使用 `java.text.umberFormat` 类来格式化数字：

```
import java.text.*;

// 使用 NumberFormat 来格式化与解析当前 locale 的数字

NumberFormat nf = NumberFormat.getNumberInstance(); // 取得 NumberFormat
System.out.println(nf.format(9876543.21)); // 针对当前 locale 来格式化数字
try {
    Number n = nf.parse("1.234.567,89"); // 依据 locale 来解析字符串
} catch (ParseException e) { /* Handle exception */ }

// 货币值的格式化有时会与其他数字不同
NumberFormat moneyFmt = NumberFormat.getCurrencyInstance();
System.out.println(moneyFmt.format(1234.56)); // 列出 U.S. 中的 $1,234.56
```

## 日期与时间

Java 允许以三种形式来表现并处理日期和时间：作为 `long` 值、作为 `java.util.Date` 或作为 `java.util.Calendar` 对象。Java 5.0 引入了 `enumerated` 类型 `java.util.concurrent.TimeUnit`。这个类型的值代表了时间的精确度或单位：秒 (`second`)、毫秒 (`millisecond`)、微秒 (`microsecond`，一百万分之一秒) 及纳秒 (`nanosecond`，十亿分之一秒)。它们具有方便有用的 `method`，但其本身不代表时间。

### 毫秒与纳秒

在最低层，日期和时间被表示为 `long` 值，这个值包括自 1970 年 1 月 1 日午夜开始的正或负毫秒数。此特殊的日期和时间也就是新纪元 (`epoch`)，并且是格林威治标准时间 (`Greenwich Mean Time, GMT`) 或世界时 (`Universal Time, UTC`)。如果要查询以此毫秒表示的当前时间，可以使用 `System.currentTimeMillis()`：



```
long now = System.currentTimeMillis();
```

在Java 5.0及之后的版本中,你可以使用`System.nanoTime()`来查询以纳秒为单位的时间。此`method`会返回纳秒长度的`long`数字。与`currentTimeMillis()`不同,`nanotime()`不会返回任何相对于已定义的新纪元的时间。`nanotime()`很适合用于测量相对或占用时间(只要占用时间不大于292年),但不适合用于绝对时间:

```
long start = System.nanoTime();
doSomething();
long end = System.nanoTime();
long elapsedNanoSeconds = end - start;
```

## Date 类

`java.util.Date`是围绕着`long`的对象封装程序,它包含了自新纪元开始的毫秒数。使用`Date`对象来代替`long`,就能用`toString` `method`进行简单的转换,以成为非本地化的字符串。`Date`对象可以用`equals()`进行相等性比较,而且可以用`compareTo()` `method`或`before()`和`after()` `method`比较顺序。

无自变量版本的`Date()`构造函数会创建一个表示当前时间的`Date`,你也可以传递毫秒的`long`数值来创建表示其他时间的`Date`。`getTime()`会返回`Date`的毫秒表达式。`Date`是可变(`mutable`)类,所以你也可以传递毫秒数给`setTime()`。

`Date`有一些用于查询和设定年、月、日、时、分和秒的`method`。但是,现在都不赞成使用这些`method`,而是支持使用`Calendar`类,如下段所述。

## Calendar 类

`java.util.Calendar`类是个适当地本地化版本的`Date`。它只是围绕于毫秒数的`long`值的封装程序,但可以表示依据当前本地的日历(通常是新历(`Gregorian calendar`))的时间。此外,它具有用于在各种日期和时间字段上查询、设定以及计算的`method`。

以下程序代码展示了`Calendar`类的常见用法。请注意,`set()`、`get()`以及`add()` `method`都会接受初始自变量,它指定了日期或时间的哪个字段要被设定、查询或加减。例如年、一月天次、一周天次、时、分和秒这样的字段,会在类中以整数常量被定义。其他的整数常量定义了新历的月和周的值,月常量`UNDECIMBER`代表阴历的第13个月。

```
// 取得当前地区和时区的Calendar
Calendar cal = Calendar.getInstance();

// 算出今天是今年的第几天
cal.setTimeInMillis(System.currentTimeMillis()); // 设定当前时间
int dayOfYear = cal.get(Calendar.DAY_OF_YEAR);    // 今天是今年的第几天?
```

```
// 2008 年的 2 月 29 日是星期几?
cal.set(2008, Calendar.FEBRUARY, 29);           // 设定年、月、日字段
int dayOfWeek = cal.get(Calendar.DAY_OF_WEEK);    // 查询另一个字段

// 2005 年 5 月的第 3 个星期四是当月的第几天?
cal.set(Calendar.YEAR, 2005);                     // 设定年
cal.set(Calendar.MONTH, Calendar.MAY);            // 设定月
cal.set(Calendar.DAY_OF_WEEK, Calendar.THURSDAY); // 设定星期几
cal.set(Calendar.DAY_OF_WEEK_IN_MONTH, 3);        // 设定周
int dayOfMonth = cal.get(Calendar.DAY_OF_MONTH);  // 查询那天是当月的第几天

// 取得表示从现在开始三个月的 Date 对象
cal.setTimeInMillis(System.currentTimeMillis()); // 当前时间
cal.add(Calendar.MONTH, 3);                        // 增加三个月
Date expiration = cal.getTime();                   // 取得 Date 形式的结果
long millis = cal.getTimeInMillis();               // 或使用 long 形式
```

## 格式化日期与时间

Date 的 toString() method 会产生文字形式的日期和时间, 但不会做本地化, 而且不允许指定显示哪些字段 (例如, 日、月和年或时、分)。toString() method 应该只用于产生机器可读取的时间, 而不是人所读取的字符串。

就和数字一样, 日期和时间可以用 String.format() method 和 Java 5.0 相关的 java.util.Formatter 类来转换为字符串。用于显示日期和时间的格式字符串都是以字母 t 开头的双字符序列。每个序列的第二个字母指定或设定了要被显示的日期或时间字段。例如 %tR 显示使用 24 小时制的时和分字段, 而 %tD 显示以斜线分隔的月、日以及年字段。String.format() 可以将指定的日期或时间格式化为 long、Date 或 Calendar:

```
// 现在的时和分
long now = System.currentTimeMillis();
String s = String.format("%tR", now);           // "15:12"

// 现在的月 / 日 / 年
Date d = new Date(now);
s = String.format("%tD", d);                     // "07/13/04"

// 使用 12 小时制的时和分
Calendar c = Calendar.getInstance();
c.setTime(d);
s = String.format("%tI:%tM %tp", now, d, c);    // "3:12 pm"
```

在 Java 5.0 及其 Formatter 类之前, 你可以用 java.text.DateFormat 类来格式化日期和时间, 它会针对日期和时间格式化并自动处理与地区相关的惯例的转换。甚至在使用了与全世界大部分地区常使用的阳历不同的日历的地区, DateFormat 也能正确处理:

```

import java.util.Date;
import java.text.*;

// 使用默认的本地格式来显示当前的日期
DateFormat defaultDate = DateFormat.getDateInstance();
System.out.println(defaultDate.format(new Date()));

// 使用本地的短时间格式来显示当前时间
DateFormat shortTime = DateFormat.getTimeInstance(DateFormat.SHORT);
System.out.println(shortTime.format(new Date()));

// 使用长格式同时显示日期和时间
DateFormat longTimestamp =
    DateFormat.getDateInstance(DateFormat.FULL, DateFormat.FULL);
System.out.println(longTimestamp.format(new Date()));

// 使用 SimpleDateFormat 来定义你自己的格式化模板 (template)
// 关于模板的语法请参阅 java.text.SimpleDateFormat
DateFormat myformat = new SimpleDateFormat("yyyy.MM.dd");
System.out.println(myformat.format(new Date()));
try {    // DateFormat 也可以解析日期
    Date leapday = myformat.parse("2000.02.29");
}
catch (ParseException e) { /* Handle parsing exception */ }

```

## 数组

java.lang.System类定义了arraycopy() method, 它对于把第一个数组中的特定元素复制到第二个数组中的指定位置很有用。第二个数组必须与第一个数组有相同的类型, 甚至可以是同一个数组:

```

char[] text = "Now is the time".toCharArray();
char[] copy = new char[100];
// 从text中的元素4开始复制10个字符到copy, 从copy[0]开始
System.arraycopy(text, 4, copy, 0, 10);

// 将text的一部分移到较后面的元素以让出地方供插入
System.arraycopy(copy, 3, copy, 6, 7);

```

在Java 1.2和之后的版本中, java.util.Arrays类定义了有用的数组操作method, 其中包括了用于排序和查找数组的method:

```

import java.util.Arrays;

int[] intarray = new int[] { 10, 5, 7, -3 }; // 整个数组
Arrays.sort(intarray);                      // 进行排序
int pos = Arrays.binarySearch(intarray, 7); // 值7是在位置2找到的
pos = Arrays.binarySearch(intarray, 12);    // 未找到: 返回负值

```



```
// 由对象组成的数组也能被排序和查找
String[] strarray = new String[] { "now", "is", "the", "time" };
Arrays.sort(strarray); // sorted to: { "is", "now", "the", "time" }

// Arrays.equals()会比较两数组中的所有元素
String[] clone = (String[]) strarray.clone();
boolean b1 = Arrays.equals(strarray, clone); // 没错, 它们相等

// Arrays.fill()会初始化数组元素
byte[] data = new byte[100]; // 空数组, 元素都被设定为0
Arrays.fill(data, (byte) -1); // 将它们都设定为-1
Arrays.fill(data, 5, 10, (byte) -2); // 将元素5, 6, 7, 8, 9设定为-2
```

在Java中, 数组可以被当成对象来处理与操作。给定任意对象`o`, 你可以使用如下的程序代码来查明此对象是否是数组, 如果是, 又是什么类型的数组:

```
Class type = o.getClass();
if (type.isArray()) {
    Class elementType = type.getComponentType();
}
```

## 集合

Java 集合结构 (Collection Framework) 是 `java.util` 包中的一组重要的实用程序类和接口, 用于处理对象的集合。Collection Framework 定义了两个基本的集合类型。`Collection` 是一组对象, 而 `Map` 是对象之间的一组映射或关联。`Set` 是一种不具有重复项目的 `Collection` 类型, 而 `List` 是其元素已被排序的 `Collection`。`SortedSet` 和 `SortedMap` 是特殊化的 `set` 和 `map`, 以排序的方式来维护它们的元素。`Collection`、`Set`、`List`、`Map`、`SortedSet` 以及 `SortedMap` 都是接口, 但 `java.util` 包也定义了各种具体的实现, 例如基于数组和链接列表的 `list` 以及基于哈希表或二元树的 `map` 和 `set`。其他的重要接口是 `Iterator` 和 `ListIterator`, 能让你逐个处理集合中的对象。Collection Framework 是在 Java 1.2 中加入的, 但在那个版本之前, 你可以使用 `Vector` 和 `Hashtable`, 它们几乎与 `ArrayList` 和 `HashMap` 相同。

在 Java 1.4 中, Collection API 增加了 `RandomAccess` 这个 marker interface, 它是由 `List` 的实现来完成, 支持了有效率的随机访问 (也就是说, 它是由 `ArrayList` 和 `Vecctor` 实现, 而不是由 `LinkedList`)。Java 1.4 也引入了 `LinkedHashMap` 和 `LinkedHashSet`, 它是基于哈希表的 `map` 和 `set`, 会保留元素的插入顺序。最后, `IdentityHashMap` 是个基于哈希表的 `Map` 实现, 它使用 `==` 运算符来比较关键对象, 而不是使用 `equals()` method 来进行比较。

Collection Framework 在 Java 5.0 中已被彻底修正为使用 generics (请参阅第四章)。Java 5.0 还增加了 `EnumSet` 和 `EnumMap` 类, 它们被特殊化以处理枚举值 (请参阅第四章) 以

及新的 `for/in` 循环语句所使用的 `java.lang.Iterable` 接口。最后, Java 5.0 增加了 `Queue` 接口。大部分令人感兴趣的 `Queue` 实现是 `java.util.concurrent` 中的 `BlockingQueue` 实现。

## Collection 接口

`Collection<E>` 是个参数化接口, 表示由类型 `E` 的对象所组成的泛型组。此组有可能允许重复元素, 也有可能对元素排序。有一些 `method` 被定义来从组增加或删除对象、测试对象在组中的成员资格以及逐一处理组中的所有元素。另有一些 `method` 会以数组形式返回组的元素并返回集合的大小。

Java Collection Framework 不提供任何 `Collection` 的实现, 但此接口仍然非常重要, 因为它定义了所有针对 `Set`、`List` 以及 `Queue` 实现的共同特性。以下程序代码说明在 `Collection` 对象上能执行的操作:

```
// 创建一些 collection 来运行
Collection<String> c = new HashSet<String>(); // 一个空的 set
// 我们稍后会看到这些实用程序 method
Collection<String> d = Arrays.asList("one", "two"); // 固定不变
Collection<String> e = Collections.singleton("three"); // 固定不变

// 增加 collection 的元素。如果 collection 变动, 这些 method 就会返回
// true, 这对于不允许重复项目的 Set 很有用
c.add("zero"); // 增加单一元素
c.addAll(d); // 增加许多元素

// 复制 collection: 大部分的实现会有个 copy 构造函数
Collection<String> copy = new ArrayList<String>(c);

// 删除 collection 的元素
// 如果 collection 有变更, 则除了 Clear() 之外, 全部都会返回 true
c.remove("zero"); // 删除一个元素
c.removeAll(e); // 删除许多元素
c.retainAll(d); // 删除所有不在 e 中的元素
c.clear(); // 删除 collection 的所有元素

// 查询 collection 的大小
boolean b = c.isEmpty(); // collection 现在是空的
int s = c.size(); // collection 的大小现在是 0

// 从我们制作的副本来复原 collection
c.addAll(copy);

// 测试 collection 中的成员数据, 成员数据是基于 equals()
// method, 而不是 == 运算符
b = c.contains("zero"); // true
b = c.containsAll(d); // true
```

```
// 以 while 循环逐一处理 collection 的元素
// 有些实现 (例如 list) 会保证处理的顺序
// 其他的则不保证
Iterator<String> iterator = c.iterator();
while(iterator.hasNext()) System.out.println(iterator.next());

// 使用 for 循环来处理
for(Iterator<String> i = c.iterator(); i.hasNext(); )
    System.out.println(i.next());

// Java 5.0 的迭代使用了 for/in 循环
for(String word : c) System.out.println(word);

// 大部分的 Collection 实现有有用的 toString() method
System.out.println(c); // 作为以上迭代的替代方案

// 取得由 Collection 元素组成的数组。如果 iterator
// 保证顺序, 此数组就会有相同顺序。此数组是个副
// 本, 不是对内部数据结构的引用
Object[] elements = c.toArray();

// 如果我们要让 String[] 中有元素, 就必须先传一个进去
String[] strings = c.toArray(new String[c.size()]);

// 或者我们可以传递一个只有指定类型的空 String[],
// 而 toArray() method 会为我们分配一个数组
strings = c.toArray(new String[0]);
```

要记住, 你可以配合任何 Set、List 或 Queue 来使用以上所示的任一个 method。这些子接口 (subinterface) 可能会影响 collection 的元素上的成员数据限制或顺序限制, 但仍提供了相同的基本 method。例如 add()、remove()、clear() 以及 retainAll() 这些会变动 collection 的 method 是选择性的, 而只读的实现可能会抛出 UnsupportedOperationException。

Collection、Map 以及其他的 subinterface 没有扩展 Cloneable 或 Serializable interface。但是, 所有 Java Collection Framework 中提供的 collection 和 map 实现类的确实现了这些 interface。

有些 collection 实现会在可包含的元素上设定限制。例如, 有个实现可能会禁止用 null 作为元素。EnumSet 将成员数据限定为属于特定枚举类型的值, 试图在 collection 中加入被禁止的元素, 一定会抛出例如 NullPointerException 或 ClassCastException 的未检查异常。检查 collection 中是否包含被禁止的元素, 也有可能抛出这样的异常或是返回 false。

## Set 接口

set 是由对象组成的 collection, 其中不允许有重复项目, 它不可以包含两个对相同对象



的引用、两个对 `null` 的引用或对两个对象 `a` 与 `b` 而且 `a.equals(b)` 的引用。大部分的通用 `Set` 实现不会对 `set` 的元素排序，但有序的 `set` 并未被禁止（请参阅 `SortedSet` 和 `LinkedHashSet`）。依据一般预期，`set` 优于像是 `list` 这样的有序 `collection`，`set` 具有一个有效率的在常量或对数时间中运行的 `contains()` `method`。

`Set` 没有定义由 `Collection` 定义的 `method` 之外的额外 `method`，但在那些 `method` 上增加了额外的限制。如果要强制执行无重复项目规则，`Set` 的 `add()` 和 `addAll()` `method` 是必要的：如果 `set` 已包含了那个元素，就不可以再把它加到 `Set` 中。回想一下，如果调用会造成 `collection` 改变，则 `Collection` `interface` 所定义的 `add()` 和 `addAll()` `method` 就会返回 `true`，否则就返回 `false`。此返回值与 `Set` 对象有关，因为无重复项目的限制表示增加元素未必一定会造成 `set` 的改变。

表 5-2 列出了 `Set` `interface` 的实现，并归纳它们的内部表示方式、排序特性、成员限制、基本 `add()`、`remove()` 和 `contains()` 运算以及迭代的性能。请注意，`CopyOnWriteArraySet` 是在 `java.util.concurrent` 包中，其他的实现都是 `java.util` 的一部分。也请注意，`java.util.BitSet` 不是一个 `Set` 实现。这个原有类在作为紧密且有效率的布尔值列表时很有用，但它不是 `Java Collection Framework` 的一部分。

表 5-2: `Set` 实现

类	内部表示方式	元素排序	成员限制	基本运作	迭代性能	注意事项
<code>HashSet</code>	哈希表	无	无	$O(1)$	$O(\text{capacity})$	最佳的通用实现
<code>LinkedHashSet</code>	链接哈希表	插入顺序	无	$O(1)$	$O(n)$	保留插入顺序
<code>EnumSet</code>	位字段	<code>enum</code> 声明	<code>enum</code> 值	$O(1)$	$O(n)$	只包含非 <code>null</code> 的 <code>enum</code> 值
<code>TreeSet</code>	red-black 树形结构	升幂排列	可比较	$O(\log(n))$	$O(n)$	可比较的元素或 <code>Comparator</code>
<code>CopyOnWriteArraySet</code>	队列	插入顺序	无	$O(n)$	$O(n)$	没有同步 <code>method</code> 的线性安全性

`TreeSet` 实现使用了树形数据结构来维护依升幂排列迭代的 `set`，排序是依据 `Comparable` 对象的自然排序或依据由 `Comparator` 对象所指定的顺序。`TreeSet` 实际上实现了 `SortedSet` 接口，那是 `Set` 的子接口。

`SortedSet` 提供了好几个令人感兴趣的 `method`，这些 `method` 运用了它的已排序本质。以下程序代码作了说明：

```
public static void testSortedSet(String[] args) {
    // 创建 SortedSet
    SortedSet<String> s = new TreeSet<String>(Arrays.asList(args));

    // 迭代 set: 元素会被自动排序
    for(String word : s) System.out.println(word);

    // 特别的元素
    String first = s.first(); // 第一个元素
    String last = s.last(); // 最后一个元素
    // set 的子范围查看
    SortedSet<String> tail = s.tailSet(first+'\0'); // 除了第一个元素之外的所有元素
    SortedSet<String> head = s.headSet(last); // 除了最后一个元素之外的所有元素
    SortedSet<String> middle = s.subSet(first+'\0', last)
    // 除了第一个元素和最后一个元素之外的所有元素
    ;
}
```

## List 接口

List 是有序的对象集合。list 的每个元素在 list 中都有特定位置，而且 List interface 定义了 method 来查询或设定特定位置（或索引）的元素。在这方面，List 就像是大小会变动的数组，会随需要来容纳它所包含的元素数量。与 set 不同的是，list 允许重复的元素。

除了基于索引的 get() 和 set() method 之外，List interface 还定义了 method 来增加或删除特定索引中的元素，也定义了 method 来返回特定值在 list 中的第一个或最后一个出现位置的索引。继承自 Collection 的 add() 和 remove() method 被定义来添加数据至 list 或从 list 删除与指定值相符的第一个元素。retainAll() 和 removeAll() method 的行为就和针对任一种 Collection 的一样，在需要时保留或删除相同的多个值。

List interface 没有定义在一段 list 索引范围内进行操作的 method，但它定义了一个 subList method，它会返回只表示原来 list 的指定范围的 List 对象。子 list 会回溯至父 list，对子 list 所做的任何改变都会立即出现在父 list 中。subList() 的范例及其他基本的 List 操作 method 如下。

```
// 创建 list 以运用
List<String> l = new ArrayList<String>(Arrays.asList(args));
List<String> words = Arrays.asList("hello", "world");

// 用索引来查询和设定元素
String first = l.get(0); // list 的第一个元素
String last = l.get(l.size()-1); // list 的最后一个元素
l.set(0, last); // 将第一个元素变为最后一个

// 增加并插入元素。add() 可以添加或插入
l.add(first); // 将 first 添加在 list 的结尾
```

```

l.add(0, first);    // 将first再次插入list的开头
l.addAll(words);    // 将一个collection添加在list的结尾
l.addAll(1, words); // 将collection插在第一个单词后面

// 子list: 回溯至原始list
List<String> sub = l.subList(1,3); // 第二及第三个元素
sub.set(0, "hi");                // 修改l的第二个元素
// subList 可以将操作限制在list的子范围内
String s = Collections.min(l.subList(0,4));
Collections.sort(l.subList(0,4));
// 子list的独立副本不会影响父list
List<String> subcopy = new ArrayList<String>(l.subList(1,3));

// 搜索list
int p = l.indexOf(last); // 最后一个单词出现在哪里?
p = l.lastIndexOf(last); // 逆向搜索

// 列出l中所有与last相符的索引。请注意 subList()
int n = l.size();
p = 0;
do {
    // 查看只包含我们尚未搜索的元素的list
    List<String> list = l.subList(p, n);
    int q = list.indexOf(last);
    if (q == -1) break;
    System.out.printf("Found '%s' at index %d%n", last, p+q);
    p += q+1;
} while(p < n);

// 从list删除元素
l.remove(last);          // 删除第一个相符的元素
l.remove(0);             // 删除特定索引中的元素
l.subList(0,2).clear();  // 使用 subList()删除一段范围内的元素
l.retainAll(words);      // 删除 words 中除了元素之外的所有东西
l.removeAll(words);      // 删除 words 中的所有元素
l.clear();               // 删除所有东西

```

对List实现的一般期待就是它们能被有效率地迭代，通常所需的时间会与list的大小成比例。但是，list并未对任一个索引的元素都提供有效率的随机访问。循序访问的列表（例如LinkedList类）以随机访问性能为代价，提供了有效率的插入和删除操作。在Java 1.4及之后的版本中，提供了有效率的随机访问的实现，包括RandomAccess这个marker interface，如果你需要确认操作的列表效率，可以用 instanceof 来测试此 interface：

```

List<?> l = ...; // 我们传递供操作的任意list
// 确定我们可以做到有效率的随机访问，否则，就在进行操作
// 前，使用 copy 构造函数建立随机访问的副本
if (!(l instanceof RandomAccess)) l = new ArrayList<?>(l);

```

由List的iterator() method所返回的Iterator，会依list元素在list中出现的顺



序来迭代。List 实现了 Iterable，而列表就像其他的 collection 一样，可以使用 for/in 循环来迭代。

如果只要迭代 list 的一部分，可以使用 subList() method 来创建子 list：

```
List<String> words = ...; // Get a list to iterate

// 迭代除了第一个元素之外的所有元素
for(String word : words.subList(1, words.size()))
    System.out.println(word);
```

除了标准的迭代之外，列表也提供强化的双向迭代，那是使用由 ListIterator() method 所返回的 ListIterator 对象。例如，如果要逆向迭代 List，可以将 ListIterator 的光标定位在列表的结尾来开始 ListIterator：

```
ListIterator<String> li = words.listIterator(words.size());
while(li.hasPrevious()) {
    System.out.println(li.previous());
}
```

表 5-1 归纳了五个在 Java 平台中的通用 List 实现。Vector 和 Stack 是 Java 1.0 留下的原有实现，CopyOnWriteArrayList 是 Java 5.0 中新出现的并且是 java.util.concurrent 包的一部分。

表 5-3: List 实现

类	代表	随机访问	备注
ArrayList	数组	是	最佳的全能实现
LinkedList	双链接列表	否	在新增和插入上有效率
CopyOnWriteArrayList	数组	是	具有线程安全性，追踪 (traversal) 时快捷，但变动时缓慢
Vector	数组	是	原有类，同步化的 method
Stack	数组	是	扩展了 Vector，增加 push()、pop()、peek()

## Map 接口

map 是由一些键对象 (key object) 和那些键对象与值对象 (value object) 的映射 (mapping) 组成的。Map interface 定义了一个用于定义和查询映射的 API。Map 是 Java Collection Framework 的一部分，但它并没有扩展 Collection interface，所以 Map 是小 c 的 collection，而不是大 C 的 Collection。Map 是具有两个类型变量的参数化类型。类型变量 K 代表 map 包含的键的类型，而类型变量 V 代表键所映射的值的类型。例如，从 String 键到 Integer 值的映射，可以用 Map<String,Integer> 来表示。

最重要的 Map method 有: `put()`, 定义了 map 中的键/值对; `get()`, 查询与特定键结合的值; `remove()`, 从 map 中删除特定键及与其关联的值。针对 Map 实现的一般性能期望就是这三个基本 method 相当有效率: 它们应该在常量时间 (constant time) 内运行, 而且不可以比在对数时间 (logarithmic time) 内的状况糟。

Map 有个重要特性, 就是它对“集合观点 (collection view)”的支持。虽然 Map 不是个 Collection, 但它的键可以被看成 Set, 它的值可以被看成 Collection, 而它的映射可以被看成由 Map.Entry 对象组成的 Set (Map.Entry 是个定义于 Map 中的嵌套 interface: 它只表示单一的键/值对)。

以下的程序代码显示了 Map 的 `get()`、`put()`、`remove()` 以及其他 method, 也说明了 Map 的 collection view 的一些常见用法:

```
// 创建 map 以运用
Map<String,Integer> m = new HashMap<String,Integer>(); // 新的空 map
// 不可改变的 Map 包含了单一键-值对
Map<String,Integer> singleton = Collections.singletonMap("testing", -1);
// 请注意, 这个很少用的语法是在明确地指定 generic emptyMap() method 的参数类型。
// 所返回的 map 是不可变更的
Map<String,Integer> empty = Collections.<String,Integer>emptyMap();

// 使用 put() 填入 map 来定义从数组元素到各个元素
// 出现位置的索引的映射
String[] words = { "this", "is", "a", "test" };
for(int i = 0; i < words.length; i++)
    m.put(words[i], i); // 请注意从 int 到 Integer 的 autoboxing

// 每个键都必须映射到一个值, 但多个键可以映射到同一个值
for(int i = 0; i < words.length; i++)
    m.put(words[i].toUpperCase(), i);

// putAll() method 会从另一个 Map 复制映射
m.putAll(singleton);

// 使用 get() method 查询映射
for(int i = 0; i < words.length; i++)
    if (m.get(words[i]) != i) throw new AssertionError();

// 键和值的成员关系测试
m.containsKey(words[0]); // true
m.containsValue(words.length); // false

// Map 的键、值以及映射项目可以被当成 collection 来查看
Set<String> keys = m.keySet();
Collection<Integer> values = m.values();
Set<Map.Entry<String,Integer>> entries = m.entrySet();

// Map 及其 collection view 通常会有有用的 toString() method
System.out.printf("Map: %s\nKeys: %s\nValues: %s\nEntries: %s\n",
    m, keys, values, entries);
```

```

// 这些 collection 可以被迭代
// 大部分的 map 都具有未定义的迭代顺序 (请参阅 SortedMap)
for(String key : m.keySet()) System.out.println(key);
for(Integer value: m.values()) System.out.println(value);

// Map.Entry<K,V> 类型代表 map 中的单一键 / 值配对
for(Map.Entry<String,Integer> pair : m.entrySet()) {
    // 列出映射
    System.out.printf("%s" ==> %d%n", pair.getKey(), pair.getValue());
    // 并递增每个项目的值
    pair.setValue(pair.getValue() + 1);
}

// 删除映射
m.put("testing", null); // 映射至 null 就可以“消除”映射
m.get("testing"); // 返回 null
m.containsKey("testing"); // 返回 true: 映射仍然存在
m.remove("testing"); // 删除映射
m.get("testing"); // 仍然返回 null
m.containsKey("testing"); // 现在返回 false

// 通过 map 的 collection view 也可以进行删除
// 但是, 不能用这个方式新增 map
m.keySet().remove(words[0]); // 与 m.remove(words[0]) 相同
m.values().remove(2); // 删除一个对值 2 的映射
m.values().removeAll(Collections.singleton(4)); // 删除所有对值 4 的映射
m.values().retainAll(Arrays.asList(2, 3)); // 只保留对 2 和 3 的映射

// 通过 iterator 也可以进行删除操作
Iterator<Map.Entry<String,Integer>> iter = m.entrySet().iterator();
while(iter.hasNext()) {
    Map.Entry<String,Integer> e = iter.next();
    if (e.getValue() == 2) iter.remove();
}

// 找出同时出现在两个 map 中的值。通常, addAll() 和 retainAll() 配合
// keySet() 和 values() 能做到并集和交集
Set<Integer> v = new HashSet<Integer>(m.values());
v.retainAll(singleton.values());

// 各种 method
m.clear(); // 删除所有的映射
m.size(); // 返回映射数量: 当前是 0
m.isEmpty(); // 返回 true
m.equals(empty); // true: Map 实现覆盖了 equals

```

Map interface 包含了各种一般用途和特殊用途的实现, 这些归纳于表 5-4。除了 ConcurrentHashMap (它是 java.util.concurrent 的一部分) 之外, 表 5-4 中的所有类都是在 java.util 包中。



表 5-4: Map 实现

类	代表	开始有 的版本	null 键	null 值	备注
HashMap	哈希表	1.2	是	是	一般用途的实现
Concurrent- HashMap	哈希表	5.0	否	否	一般用途、具有线程安全性的实现。 请参阅 ConcurrentMap interface
EnumMap	array	5.0	否	是	键是 enum 的 instance
LinkedHashMap	哈希表 加上列表	1.4	是	是	会保存插入或访问的顺序
TreeMap	red-black 树形结构	1.2	否	是	依键值排序。操作为 $O(\log(n))$ 。请参阅 SortedMap
IdentityHashMap	哈希表	1.4	是	是	使用 == 来比较，而不是用 equals()
WeakHashMap	哈希表	1.2	是	是	不会阻止键的内存回收
Hashtable	哈希表	1.0	否	否	原有类；同步化的 method
Properties	哈希表	1.0	否	否	以 String method 扩展 Hashtable

java.util.concurrent包的ConcurrentHashMap类实现了同一个包的ConcurrentMap interface。ConcurrentMap扩展了Map并定义了一些额外的基本操作，这些在多线程程序设计中是很重要的。例如，putIfAbsent() method就像是put()，但只有在键尚未被映射时才会对map增加键/值对。

TreeMap实现了SortedMap interface，它扩展了Map来加入能运用map的排序本质的method。SortedMap与SortedSet interface相当类似。firstKey()和lastKey() method会返回keySet()中的第一个和最后一个键，而headMap()、tailMap()和subMap()会返回限定范围的原始map。

## Queue 与 BlockingQueue 接口

队列 (queue) 是具有顺序性的元素集合，包含从队列的开头依序提取元素的method。队列的实现通常是基于如先进先出 (FIFO) 队列或后进先出队列 (LIFO，队列也就是堆栈 (stack)) 的插入顺序。但是，其他的顺序也是有可能的：优先权队列 (priority queue) 会依外部的 Comparator 对象或依 Comparable 元素的天然顺序来对其元素排序。与 Set 不同的是，Queue 的实现通常会允许重复的元素。而与 List 不同的是，Queue interface 没有定义用于操作任意位置的队列元素的method，只有在队列前端的元素才能被查看。对于 Queue 实现来说，拥有固定的容量是很常见的：当队列已满时，就不能再加入元素。同样地，当队列是空的，就不能再移除任何元素。因为满和空的状况是许多

以队列为基础的算法的基础部分，所以 Queue interface 定义了一些 method，以返回值来表示这些状况，而不是抛出异常。尤其是 peek() 和 poll() method 会返回 null 来表示队列是空的。基于此原因，大部分的 Queue 实现不允许 null 元素。

阻塞型队列 (blocking queue) 是一种定义了具阻塞性的 put() 和 take() method 的队列。put() method 会增加一个元素到队列，并在必要时等待，直到队列中有空间提供给此元素。而 take() method 会从队列的前端移除一个元素，并在必要时等待，直到有元素可供移除。阻塞型队列是许多线程算法的重要部分，而 BlockingQueue interface (扩展了 Queue) 被定义为 java.util.concurrent 包的一部分。Queue、BlockingQueue 以及它们的实现是 Java 5.0 中新出现的。关于 BlockingQueue 实现的列表，请参阅本章稍后的“阻塞型队列”一节。

除了特定的多线程程序设计风格之外，队列在使用上远不如 set、list 以及 map。我们在这不使用程序代码范例，而是试着厘清大量令人困惑的队列插入和移除操作：

- 增加元素至队列

add()

此 Collection method 只是以一般方式来增加元素。在有限队列中，如果队列已满，则此 method 可能会抛出异常。

offer()

此 Queue method 就像 add()，但在元素因为有限队列已满而无法被加入时就会返回 false，而不是抛出异常。

BlockingQueue 定义了超时版本的 offer()，它会等待特定的时间，以在已满的队列中得到空间。就和基本版本的 method 一样，如果元素已被插入，就返回 true，否则返回 false。

put()

此 BlockingQueue method 会阻塞：如果元素因为队列已满而无法被插入，put() 就会等待，直到有其他的线程从队列中移除一个元素，使得新元素有空间可用为止。

- 从队列移除元素

remove()

除了 Collection.remove() method 之外，它会从队列中移除特定的元素。Queue interface 定义了无自变量版本的 remove()，会移除并返回在队列前端的元素。如果队列是空的，则此 method 会抛出 NoSuchElementException。

`poll()`

此 Queue method 会移除并返回在队列前端的元素，就和 `remove()` 所做的一样，但如果队列是空的，就返回 `null`，而不是抛出异常。

`BlockingQueue` 定义了超时版本的 `poll()`，它会针对元素被加入空队列而等待特定时间。

`take()`

此 `BlockingQueue` method 会移除并返回队列前端的元素。如果队列是空的，它就会停止，直到有其他的线程增加一个元素到此队列为止。

`drainTo()`

此 `BlockingQueue` method 会移除队列中的所有元素，并把它们加到指定的 `Collection` 中。它不会停下来等待要被加到队列中的元素。此 method 的变体会接收最大数量的元素并移出。

- 查询前端的元素，而不将它从队列中删除

`element()`

此 Queue method 会返回队列前端的元素，但不会把那个元素从队列删除。如果队列是空的，它就会抛出 `NoSuchElementException`。

`peek()`

此 Queue method 就像 `element()`，但如果队列是空的，就会返回 `null`。

在 Java 5.0 中，`LinkedList` 类已被更新来实现 Queue。它提供了无限制的 FIFO (first in, first out) 顺序，而且插入和删除只需要常量时间。`LinkedList` 允许 `null` 元素，但在 list 被用来作为队列时，最好不要使用 `null` 元素。

`java.util` 包的另一个 Queue 实现是 `PriorityQueue`，它是依据 `Comparator` 来排序其元素，或是依据由 `compareTo()` method 所定义的顺序来排序 `Comparable` 元素。`PriorityQueue` 的前端总是依据已定义顺序所列出的最小元素。

`java.util.concurrent` 包包含了一些 `BlockingQueue` 实现，在本章稍后会作说明。此包也包含了 `ConcurrentLinkedQueue`，那是个有效率的具有线程安全性的 Queue 实现，不必有同步化 method 的额外耗费。

## Collection 封装程序

对于相当多的设计出来配合 `collection` 使用的静态实用程序 method，`java.util.Collections` 类是其发源地。在这些 method 中，有一组很重要的就是 `collection` 的 `wrapper` method：它们会返回特殊用途的 `collection`，包含你所指定的



collection。wrapper collection 的用途是将额外的功能封装至本身未提供那些功能的 collection 中。wrapper 是用来提供线程安全性、写保护和运行时类型检查。wrapper collection 一定会由原始的 collection 支持，意味着 wrapper 的 method 只是发送被封装 collection 的相等 method。这表示 wrapper 对 collection 所做的改动，可以通过被封装的 collection 看到，反之亦然。

第一组 wrapper method 提供了在 collection 之外具有线程安全性的 wrapper。除了原有的 Vector 和 Hashtable 类之外，java.util 中的 collection 实现不具有同步化的 method，而且无法避免多个线程的并行访问。如果你需要具有线程安全性的 collection，可以用如下的程序代码来创建：

```
List<String> list = Collections.synchronizedList(new ArrayList<String>());
Set<Integer> set = Collections.synchronizedSet(new HashSet<Integer>());
Map<String,Integer> map =
    Collections.synchronizedMap(new HashMap<String,Integer>());
```

第二组 wrapper method 提供了 collection 对象，底层的 collection 无法通过这些对象而被修改。它们会返回只读的 collection 以供查找：任何尝试改变 collection 的内容，都会造成 UnsupportedOperationException。当你传递 collection 给 method 而且无论如何都不允许 collection 的内容被变动或改变时，这些 wrapper 会很有用：

```
List<Integer> primes = new ArrayList<Integer>();
List<Integer> readonly = Collections.unmodifiableList(primes);
// 我们可以通过 primes 修改 list
primes.addAll(Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19));
// 但我们无法通过只读的 wrapper 进行修改
readonly.add(23); // UnsupportedOperationException
```

最后一组 wrapper method 提供了被加到 collection 的值的运行时类型检查。这是 Java 5.0 中增加的，以补充由 generics 所提供的编译期类型安全性。在处理还没被转换成使用 generics 的原有程序代码时，这些 wrapper 会很有用。例如，如果你有个 SortedSet<String>，而且必须把其传递给预期有 Set 的 method 时，就可以使用检验 wrapper 来保证那个 method 不能增加任何东西到不是 String 的 set 上：

```
SortedSet<String> words = new TreeSet<String>(); // 一个 set
SortedSet<String> checkedWords = // 一个检验 set
    Collections.checkedSortedSet(words, String.class);
addWordsFromFile(checkedWords, filename); // 传递给原有 method
```

## 特殊的 Collections

除了 wrapper method 之外，java.util.Collections 类也定义了实用程序 method，用于创建不可变的 collection instance，其中包含了单一元素和其他用于创建空 collection

的 `method`。`singleton()`、`singletonList()`以及`singletonMap()`会返回不可变的 `Set`、`List` 和 `Map` 对象，其中包含单一赋值对象或单一键/值对。这些 `method` 很有用，例如，当你必须传递单一对象到预期要有 `collection` 的 `method` 时。

`Collections`类也包含了会返回空`collection`的`method`。如果你编写了会返回`collection`的`method`，通过返回空`collection`而不是返回像`null`这样的特殊值来处理无值返回(`no-values-to-return`) 情况通常是最好方法：

```
Set<Integer> si = Collections.emptySet();
List<String> ss = Collections.emptyList();
Map<String,Integer> m = Collections.emptyMap();
```

最后，`nCopies()`会返回不可变的 `List`，其中包含单一特定对象的特定数量的副本：

```
List<Integer> tenzeros = Collections.nCopies(10, 0);
```

## 与数组互相转换

由对象和`collection`所组成的数组是用于类似的用途。`list`与数组两者之间互相转换是有可能的：

```
String[] a = { "this", "is", "a", "test" }; // 一个数组
List<String> l = Arrays.asList(a); // 将数组视为不可成长的 list
List<String> m = new ArrayList<String>(l); // 创建可成长的查看副本

// 在Java 5.0中，asList()是个varargs method，所以我们可以这么做：
Set<Character> abc = new HashSet<Character>(Arrays.asList('a', 'b', 'c'));

// 定义了toArray() method。此无自变量的版本创建了一个Object[]数组，将
// 元素复制过来并将其返回
Object[] members = set.toArray(); // 让 set 元素成为数组

Object[] items = list.toArray(); // 让 list 元素成为数组
Object[] keys = map.keySet().toArray(); // 让 map 键对象成为数组
Object[] values = map.values().toArray(); // 让 map 值对象成为数组

// 如果你想让返回值是Object[]以外的东西，可以传入由适当类型组成的
// 数组。如果数组太大，那么被复制到数组的 collection 元素就会以 null
// 结束
String[] c = l.toArray(new String[0]);
```

## Collections 实用程序 method

`java.util.Arrays`类定义了`method`来操作数组，`java.util.Collections`类也定义了`method`来操作`collection`。最值得注意就是排序和搜索`collection`元素的`method`：

```
Collections.sort(list);
int pos = Collections.binarySearch(list, "key"); // list 必须已被排序
```

这有一些令人感兴趣的其他 Collection method:

```
Collections.copy(list1, list2); // 将list2复制到list1, 改写list1
Collections.fill(list, o);      // 以Object o填充list
Collections.max(c);             // 找出Collection c中的最大元素
Collections.min(c);             // 找出Collection c中的最小元素

Collections.reverse(list);      // 反转list
Collections.shuffle(list);      // 混合list
```

## 实现 Collection

Java Collections Framework提供了抽象类,使得实现常用的collection的类型变得简单。以下的程序代码扩展AbstractList来定义QuadraticSequence。这是个列表实现,它会依需求计算列表值,而不是把它们实际存放在内存中的某处。请参阅AbstractSet、AbstractMap、AbstractQueue以及AbstractSequentialList。

```
import java.util.*;

/** 不可变的List<Double>表示序列  $ax^2 + bx + c$  */
public class QuadraticSequence extends AbstractList<Double> {
    final int size;
    final double a, b, c;

    QuadraticSequence(double a, double b, double c, int size) {
        this.a = a; this.b = b; this.c = c; this.size = size;
    }

    @Override public int size() { return size; }

    @Override public Double get(int index) {
        if (index < 0 || index >= size) throw new ArrayIndexOutOfBoundsException();
        return a * index * index + b * index + c;
    }
}
```

## 线程与并行

Java平台从Java 1.0开始就以Thread类和Runnable接口来支持多线程或并行程序设计。Java 5.0以针对并行程序设计的新的实用程序的广泛组来加强支持。

### 创建、运行以及操作线程

Java使得在程序中定义与运作多个线程变得容易。java.lang.Thread是Java API中的基础线程类。定义线程的方法有两种:一种是制作Thread的子类,覆盖run() method,然后实例化你的Thread子类;另一种是定义实现了Runnable method的类



(也就是定义 `run()` method), 然后传递此 `Runnable` 对象的 instance 给 `Thread()` 构造函数。无论是哪种方法, 结果都是 `Thread` 对象, 其中 `run()` method 是线程的主体。当你调用 `Thread` 对象的 `start()` method 时, 解释器会创建一个新的线程来执行 `run()` method。此新线程会继续运行, 直到 `run()` method 结束。此时, 原来的线程会从接在 `start()` method 后面的语句开始继续运行其本身。以下程序代码说明这点:

```
final List list; // 某些长的未排序的对象列表; 已在其他地方初始化

/** 用来在后台对 List 排序的 Thread 类 */
class BackgroundSorter extends Thread {
    List l;
    public BackgroundSorter(List l) { this.l = l; } // 构造函数
    public void run() { Collections.sort(l); } // 线程主体
}

// 创建 BackgroundSorter 线程
Thread sorter = new BackgroundSorter(list);
// 开始运行: 在原来的线程继续执行接下来
// 的语句时, 新的线程会执行 run() method
sorter.start();

// 这是另一个定义类似线程的方法
Thread t = new Thread(new Runnable() { // 创建一个新的线程
    public void run() { Collections.sort(list); } // 为对象列表排序
});
t.start(); // 开始运行
```

## 线程的生命周期

线程可以处于六种状态中的一种。在 Java 5.0 中, 这些状态是由 `Thread.State` 枚举类型表示, 而线程的状态可以用 `getState()` method 来查询。`Thread.State` 常量列表很好地提供了线程生命周期的一览:

### NEW

`Thread` 已被创建, 但其 `start()` method 尚未被调用。所有线程都会从这个状态开始。

### RUNNABLE

线程正在运行或在操作系统调度它时就可运行。

### BLOCKED

因为线程在等待取得锁定以便进入同步 method 或程序块, 所以线程并未运行。我们会在本节的稍后看到更多关于同步 method 和程序块的信息。

### WAITING

线程因为调用了 `Object.wait()` 或 `Thread.join()` 而未运行。

## TIMED\_WAITING

线程因为调用了 `Thread.sleep()` 或加上逾时值来调用 `Object.wait()` 或 `Thread.join()` 而未运行。

## TERMINATED

线程已运行完毕。它的 `run()` method 已正常结束或通过抛出异常而结束。

## 线程优先级

线程可以以不同的优先级运行。指定优先级的线程通常会在已无具较高优先级的线程等待时才会运行。在运作线程优先级时，你可以使用这里的一些程序代码：

```
// 将线程优先级设定为低于一般标准
t.setPriority(Thread.NORM_PRIORITY-1);

// 将线程的优先级设定为低于当前线程
t.setPriority(Thread.currentThread().getPriority() - 1);

// 不需等待 I/O 的线程应该要明确地让出 CPU，以让其他具有相同优先
// 级的线程有机会运行
Thread t = new Thread(new Runnable() {
    public void run() {
        for(int i = 0; i < data.length; i++) { // 逐一处理一组数据
            process(data[i]);                // 加以处理
            if ((i % 10) == 0)                // 但在每处理 10 个后，就暂
                Thread.yield();              // 停以让其他线程运行
        }
    }
});
```

## 处理未被捕获的异常

线程通常会在到达 `run()` method 终点或执行到那个 method 中的 `return` 语句时终止。但是，线程也可以通过抛出异常来终止。当线程以这种方式结束时，默认的行为就是列出线程的名称、异常类型、异常信息以及堆栈追踪。在 Java 5.0 中，你可以对线程中未被捕获的异常安装自定义的处理程序。例如：

```
// 此线程正好抛出一个异常
Thread t = new Thread() {
    public void run() {throw new UnsupportedOperationException();}
};

// 给线程一个名称以帮助调试
t.setName("My Broken Thread");

// 这是针对错误的处理程序
t.setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {
    public void uncaughtException(Thread t, Throwable e) {
```

```

        System.err.printf("Exception in thread %d '%s':" +
            "%s at line %d of %s%n",
            t.getId(), // 线程 id
            t.getName(), // 线程名称
            e.toString(), // 异常名称与信息
            e.getStackTrace()[0].getLineNumber(), // 行号
            e.getStackTrace()[0].getFileName()); // 文件名
    }
});

```

## 使线程休眠

通常，线程被用来执行一些固定间隔时间的重复工作。在进行牵涉到动画或类似效果的图形程序设计时更是如此。做到这个操作的关键就是让线程在特定的一段时间休眠或停止运行。这可以用静态的 `Thread.sleep()` method 做到，或者在 Java 5.0 中可以用 `TimeUnit` 类的列举常量的实用程序 method：

```

import static java.util.concurrent.TimeUnit.SECONDS; // 实用程序类

public class Clock extends Thread {
    // 此字段是易变的，因为两个不同的线程会访问它
    volatile boolean keepRunning = true;

    public Clock() { // 构造函数
        setDaemon(true); // daemon 线程：解释器可以在它运行时跳出
    }

    public void run() { // 线程主体
        while(keepRunning) { // 此线程会运行直到被要求停止为止
            long now = System.currentTimeMillis(); // 取得当前时间
            System.out.printf("%ttr%n", now); // 输出当前时间
            try { Thread.sleep(1000); } // 等待 1000 毫秒
            catch (InterruptedException e) { return; } // 在中断时结束
        }
    }

    // 要求线程停止运行。这是 interrupt() 的替代方案
    public void pleaseStop() { keepRunning = false; }

    // 此 method 示范如何使用 Clock 类
    public static void main(String[] args) {
        Clock c = new Clock(); // 创建 Clock 线程
        c.start(); // 激活它
        try { SECONDS.sleep(10); } // 等待 10 秒
        catch (InterruptedException ignore) {} // 忽略中断
        // 现在停止 clock thread。我们也可以使用 c.interrupt()
        c.pleaseStop();
    }
}

```

请注意此范例中的 `pleaseStop()` method：它被设计来以可控制的方式来停止时钟线



程。此范例被编写为可以通过调用从 Thread 继承来的 interrupt() method 来停止。Thread 类定义了 stop() method, 但不建议使用。

## 运行与安排任务

Java 提供了一些方法, 可以在无需显式地创建 Thread 对象的情况下, 异步地执行任务或安排它们以供将来执行。接下来的章节介绍 Java 1.3 中增加的 Timer 类和 Java 5.0 的 java.util.concurrent 包的执行者 (executor) 框架。

### 使用 Timer 安排工作

在 Java 1.3 中加入的 java.util.Timer 和 java.util.TimerTask 类使得重复任务的执行变得容易。这有一些程序代码的行为与先前显示的 Clock 类非常像:

```
import java.util.*;

// 定义显示时间的任务
TimerTask displayTime = new TimerTask() {
    public void run() { System.out.printf("%tr%n",
        System.currentTimeMillis()); }
};

// 创建 timer 对象来执行此任务 (或其他任务)
Timer timer = new Timer();
// 现在将那个任务定为每 1000 毫秒执行一次, 就从现在开始
timer.schedule(displayTime, 0, 1000);

// 停止显示时间的任务
displayTime.cancel();
```

### Executor 接口

在 Java 5.0 中, java.util.concurrent 包包含了 Executor interface。Executor 是个可以执行 Runnable 对象的对象。Executor 的使用者通常不需要知道 Executor 完成工作的方法: 它只需要知道 Runnable 会在某个时间点执行。创建 Executor 实现可以使用一些不同的线程策略, 从以下程序代码中可较清楚地得知 (请注意, 此范例也示范了 BlockingQueue 的用法)。

```
import java.util.concurrent.*;

/** 在当前线程中执行 Runnable。 */
class CurrentThreadExecutor implements Executor {
    public void execute(Runnable r) { r.run(); }
}

/** 使用新创建的线程来执行各个 Runnable */
class NewThreadExecutor implements Executor {
```

```

        public void execute(Runnable r) { new Thread(r).start(); }
    }

    /**
     * 创建一个线程来排列 Runnable 并依序加以执行
     */
    class SingleThreadExecutor extends Thread implements Executor {
        BlockingQueue<Runnable> q = new LinkedBlockingQueue<Runnable>();
        public void execute(Runnable r) {
            // 这里不执行 Runnable, 只是把它放入队列
            // 我们的队列是不受限的, 所以应该不会阻塞
            // 由于它不会阻塞, 所以绝不会抛出 InterruptedException
            try { q.put(r); }
            catch(InterruptedException never) { throw new AssertionError(never); }
        }

        // 这是实际执行了 Runnable 的线程主体
        public void run() {
            for(;;) { // 无限循环
                try {
                    Runnable r = q.take(); // 取得下一个 Runnable 或等待
                    r.run(); // 运行!
                }
                catch(InterruptedException e) {
                    // 如果被中断, 就停止执行已排入队列的 Runnable
                    return;
                }
            }
        }
    }
}

```

这些示例实现帮助说明 Executor 运作的方法以及它分隔由安排策略执行任务及线程实现细节的方法。但是, 很少有必要真的实现你自己的 `Executor`, 因为 `java.util.concurrent` 提供了有灵活性且强大的 `ThreadPoolExecutor` 类。此类通常会通过 `Executor` 类中的静态 `factory method` 来使用:

```

Executor oneThread = Executors.newSingleThreadExecutor(); // 1 的池的大小
Executor fixedPool = Executors.newFixedThreadPool(10); // 池中有 10 个线程
Executor unboundedPool = Executors.newCachedThreadPool(); // 要多少有多少

```

除了这些方便的 `factory method` 之外, 如果你想指定线程池的最小与最大的大小或想针对不会被线程立即执行的任务来指定队列类型 (例如有限制的、无限制的、依优先级排序或同步化) 以供使用, 也可以显式地创建 `ThreadPoolExecutor`。

## ExecutorService

如果你查看了 `ThreadPoolExecutor` 或以上 `Executor factory method` 所引用的签名, 就会知道它是个 `ExecutorService`。 `ExecutorService interface` 扩展了 `Executor`, 并加上了执行 `Callable` 对象的能力。 `Callable` 与 `Runnable` 很像, 但是 `Callable` 不

是随意把程序代码封装在 `run()` method 中, 而是把程序代码放在 `call()` method 中。`call()` 与 `run()` 有两个地方非常不同: 它会返回一个结果, 而且允许抛出异常。

因为 `call()` 会返回一个结果, 所以 `Callable` interface 会接收结果的类型作为参数。例如, 计算大质数的耗时程序块可以被封装在 `Callable<BigInteger>` 中:

```
import java.util.concurrent.*;
import java.math.BigInteger;
import java.util.Random;
import java.security.SecureRandom;

/** 这是针对计算大质数的 Callable 实现 */
public class RandomPrimeSearch implements Callable<BigInteger> {
    static Random prng = new SecureRandom(); // self-seeding
    int n;
    public RandomPrimeSearch(int bitsize) { n = bitsize ; }
    public BigInteger call() { return BigInteger.probablePrime(n, prng); }
}
```

当然, 你可以直接调用任一个 `Callable` 对象的 `call()` method, 但如果要执行, 则要用 `ExecutorService` 把它传递给 `submit()` method。因为 `ExecutorService` 实现通常会以异步方式执行任务, 所以 `submit()` method 不可以只返回 `call()` method 的结果。`submit()` method 会返回 `Future` 对象。`Future` 只是对未来某个时间的结果承诺, 它是对结果的类型来参数化, 如以下程序代码片段所示:

```
// 尝试同时计算两个质数
ExecutorService threadpool = Executors.newFixedThreadPool(2);
Future<BigInteger> p = threadpool.submit(new RandomPrimeSearch(512));
Future<BigInteger> q = threadpool.submit(new RandomPrimeSearch(512));
```

一旦你有了 `Future` 对象, 能用来做什么事呢? 你可以调用 `isDone()` 来查看 `Callable` 是否已完成执行; 你可以调用 `cancel()` 来取消 `Callable` 的执行, 而且可以调用 `isCancelled()` 来得知 `Callable` 是否在完成前被取消。但大部分时候, 你只会调用 `get()` 来取得 `call()` method 的结果。如果有必要, `get()` 程序块会等待 `call()` method 完成。以下程序代码可以用来配合上面显示的 `Future` 对象使用:

```
BigInteger product = p.get().multiply(q.get());
```

请注意, `get()` method 可能会抛出 `ExecutionException`。应记住 `Callable.call()` 可以抛出任何种类的异常。如果这个状况发生了, `Future` 就会把那个异常封装在 `ExecutionException` 中并从 `get()` 将它抛出。请注意, `Future.isDone()` method 会把 `Callable` 想成是“已完成 (done)”, 即使 `call()` method 是因为有异常而不正常终止。



## ScheduledExecutorService

ScheduledExecutorService 是 ExecutorService 的扩展，它增加了类似 Timer 的进度安排能力。它能让你将 Runnable 或 Callable 安排为延迟一段指定时间之后执行或把 Runnable 安排为重复执行。在各种情况下，针对未来执行的进度安排工作的结果就是 ScheduledFuture 对象。Future 也实现了 Delay interface 并提供了 getDelay() method，可以用来查询在任务开始之前的剩余时间。

取得 ScheduledExecutorService 的最简单方法就是使用 Executor 类的 factory method。以下程序代码使用 ScheduledExecutorService 来重复执行一个动作，也在固定时间间隔之后取消重复的动作。

```
/**
 * 以每秒 cps 个字符的速度输出随机 ASCII 字符，总
 * 共持续 totalSeconds 秒
 */
public static void spew(int cps, int totalSeconds) {
    final Random rng = new Random(System.currentTimeMillis());
    final ScheduledExecutorService executor =
        Executors.newSingleThreadScheduledExecutor();
    final ScheduledFuture<?> spewer =
        executor.scheduleAtFixedRate(new Runnable() {
            public void run() {
                System.out.print((char) (rng.nextInt('~' - ' ') + ' '));
                System.out.flush();
            }
        },
                                0, 1000000/cps, TimeUnit.MICROSECONDS);
    executor.schedule(new Runnable() {
        public void run() {
            spewer.cancel(false);
            executor.shutdown();
            System.out.println();
        }
    },
                                totalSeconds, TimeUnit.SECONDS);
}
```

## 互斥与锁

在使用多线程时，如果你允许多个线程访问同一个数据结构，就必须非常小心。考虑一下，当一个线程正试着逐一处理 List 中的元素时，而另一个线程正在排序那些元素，这样会发生什么事。预防这类有害的并行操作是多线程计算的主要问题之一。避免两个线程同时访问同一个对象的基本技巧，就是要求线程必须先取得对象的锁，才能加以修改。当任一个线程占有锁时，另一个请求得到锁的线程就必须等待，直到第一个线程完成任务并释放锁。每个 Java 对象都有基本功能来提供这样的锁定能力。

要让对象具有线程安全性的最简单方式就是把所有具有敏感性的 method 声明为 synchronized。线程必须取得对象的锁才可以执行它的 synchronized method，这代表其他的线程都不可以同时执行任何其他的 synchronized method（如果 static method 被声明为 synchronized，线程就必须取得类的锁且以相同的方式运作）。如果要做到较精细的锁定，你可以指定在短时间内占有指定对象的锁的 synchronized 程序代码块：

```
// 此 method 在 synchronized 块中交换两个数组元素
public static void swap(Object[] array, int index1, int index2) {
    synchronized(array) {
        Object tmp = array[index1];
        array[index1] = array[index2];
        array[index2] = tmp;
    }
}

// java.util 中的 Collection、Set、List 和 Map 的实现不具
// 有 synchronized method（除了原有的 Vector 和 Hashtable
// 实现）。在处理多线程时，可以取得同步的封装程序
// 的对象

List synclist = Collections.synchronizedList(list);
Map synclist = Collections.synchronizedMap(map);
```

## java.util.concurrent.locks 包

请注意，当你使用 synchronized 修饰符或语句时，你所要求的锁是作用在块内，当线程离开 method 或块时，它就会被自动释放。Java 5.0 中的 java.util.concurrent.locks 包提供了替代方案：你可以用 Lock 对象显式地锁定与开锁。Lock 对象不会自动以块为作用范围，你必须小心使用 try/finally 结构来确保锁一定会被释放。另一方面，Lock 使得只运用以块为作用范围的锁无法达到的算法成为可能，例如以下的“交叉前进（hand-over-hand）”链接列表追踪：

```
import java.util.concurrent.locks.*; // New in Java 5.0

/**
 * 由类型 E 的值所组成的链接列表的部分实现
 * 它示范了以 Lock 做交叉前进锁定
 */
public class LinkedList<E> {
    E value; // 列表中 this 节点的值
    LinkedList<E> rest; // 列表的其他部分
    Lock lock; // this 节点的锁

    public LinkedList(E value) { // 列表的构造函数
        this.value = value; // 节点值
        rest = null; // 这是列表中的唯一节点
        lock = new ReentrantLock(); // 我们可以锁定 this 节点
    }
}
```

```

/**
 * 附加一个节点到列表末端，使用交叉前进锁定来追踪列表。
 * 此method是具有线程安全性的：多个线程可以同时追踪
 * 列表的不同部分
 */
public void append(E value) {
    LinkedList<E> node = this; // 从this节点开始
    node.lock.lock();         // 锁定它

    // 循环处理，直到发现列表中的最后一个节点
    while(node.rest != null) {
        LinkedList<E> next = node.rest;

        // 这是交叉前进的部分。锁定下一个节点，然后释放
        // 当前节点的锁。我们使用try/finally结构，以
        // 让当前节点即使在对下一个节点的锁定因为异常
        // 而失败时仍能释放锁

        try { next.lock.lock(); } // 锁定下一个节点
        finally { node.lock.unlock(); } // 释放当前节点的锁
        node = next;
    }

    // 在这时，此节点是列表中的最后一个节点，我们
    // 对它做锁定。使用try/finally来确保释放锁
    try {
        node.rest = new LinkedList<E>(value); // 添加新节点
    }
    finally { node.lock.unlock(); }
}
}

```

## 死锁

当你使用锁定来预防线程同时访问同一个数据时，就必须小心避免死锁，这会在当两个线程互相等待对方释放它们所需的锁时。由于两者都无法继续，也没有任何一方会释放所占有的锁，所以它们都会停止运行。以下程序代码有发生死锁的倾向。死锁是否发生会随系统而异，也会随每次执行而异。

```

// 当两个线程试图锁定两个对象时，死锁就能发生，除
// 非它们都是以相同顺序要求锁
final Object resource1 = new Object(); // 这有两个要锁定的对象
final Object resource2 = new Object();
Thread t1 = new Thread(new Runnable() { // 先锁定 resource1，然后锁定 resource2
    public void run() {
        synchronized(resource1) {
            synchronized(resource2) { compute(); }
        }
    }
});

```



```
Thread t2 = new Thread(new Runnable() { // 先锁定 resource2, 然后锁定 resource1
    public void run() {
        synchronized(resource2) {
            synchronized(resource1) { compute(); }
        }
    }
});

t1.start(); // 锁定 resource1
t2.start(); // 锁定 resource2, 现在两个线程都无法进行!
```

## 协调线程

在多线程程序设计中, 要求一个线程等待另一个线程来采取某些操作是很常见的。Java 平台提供了一些方法来协调线程, 其中包括了内置于 `Object` 和 `Thread` 类的 `method` 以及 Java 5.0 中引入的 “synchronizer” 实用程序类。

### wait()与 notify()

有时线程必须停止运行并等待, 直到某个事件发生, 在那之后它会被告知继续运行。这可以用 `wait()` 和 `notify()` `method` 来实现。但是, 这些不是 `Thread` 类的 `method`, 它们是 `Object` 的 `method`。就和每个 Java 对象都有与其相关联的锁一样, 每个对象都可以维持等待线程列表。当线程调用对象的 `wait()` `method` 时, 线程所占有的每一个锁都会被暂时释放, 而线程会被加到那个对象的等待线程列表并停止运行。当另一个线程调用同一个对象的 `notifyAll()` `method` 时, 对象就会唤醒等待的线程并允许它们继续运行:

```
import java.util.*;

/**
 * 一个队列。有个线程会调用 push() 来把一个对象插入队列。
 * 另一个线程会调用 pop() 将对象从队列取出。如果没有数据, pop() 就会等待, 直到有数据为止。这里所使用的是 wait()/notify()。
 * wait() 和 notify() 必须被使用在 synchronized method 或程序块中。在 Java 5.0 中, 要用 java.util.concurrent.BlockingQueue 来代替
 */
public class WaitingQueue<E> {
    LinkedList<E> q = new LinkedList<E>(); // 存储对象的地方
    public synchronized void push(E o) {
        q.add(o); // 将对象添加到列表的末尾
        this.notifyAll(); // 告诉等待中的线程, 数据已准备好
    }
    public synchronized E pop() {
        while(q.size() == 0) {
            try { this.wait(); }
            catch (InterruptedException ignore) {}
        }
    }
}
```

```

        return q.remove(0);
    }
}

```

请注意，这样的类在 Java 5.0 中是不必要的，因为 `java.util.concurrent` 定义了 `BlockingQueue interface` 和例如 `ArrayBlockingQueue` 这样的一般用途的实现。

## 等待指定条件

Java 5.0 为对象的 `wait()` 和 `notifyAll()` method 提供了替代方案。`java.util.concurrent.locks` 定义了具有 `await()` 和 `signalAll()` method 的 `Condition` 对象。`Condition` 对象一定会与 `Lock` 对象相结合，而且在用法上与置于每个 Java 对象内的锁定和等待能力大都相同。它的主要用途就是让每个 `Lock` 具有多个 `Condition` 成为可能，这在使用基于对象的锁定和等待时是不可能的。

## 等待线程完成

有时一个线程必须停止并等待另一个线程完成。你可以用 `join()` method 来完成：

```

List list; // 要被排序的详细列表，已在其他地方初始化

// 定义线程来排序列表：降低其优先级，以让它只
// 有在当前线程在等待 I/O 时才开始运行
Thread sorter = new BackgroundSorter(list); // 先前已定义
sorter.setPriority(Thread.currentThread.getPriority()-1); // 降低优先级
sorter.start(); // 开始排序

// 同时，在原先的线程中从文件中读取数据
byte[] data = readData(); // 在其他地方定义的 method

// 在能继续处理之前，列表需已被彻底排序，所以如果 sorter 线程
// 还没完成，则我们必须等它结束
try { sorter.join(); } catch (InterruptedException e) {}

```

## 同步化实用程序

`java.util.concurrent` 包含了四个“同步化程序 (synchronizer)”类，能通过让线程等待直到出现指定条件为止来同步化并程序的状态：

### Semaphore

`Semaphore` 类模拟了信号量 (semaphore)，是传统的并行程序设计结构。概念上，`semaphore` 代表了一个或多个“许可证” (permit)。需要许可证的线程会调用 `acquire()`，接着在使用完时调用 `release()`。如果没有可用的许可证，`acquire()` 就会停止以使线程暂停，直到另一个线程释放许可证为止。

### CountDownLatch

CountDownLatch 在概念上是任一个具有两种可能状态而且从初始状态到最终状态只会变换一次的变量或并行结构。一旦变换发生，就会永久保持在最终状态。CountDownLatch 是个并行实用程序，它可以存在两种状态：关闭与开启。在初始的关闭状态中，调用 `await()` method 的线程会暂停而且不能继续处理，直到变换为开启状态为止。一旦此变换发生，所有等待中的线程就会继续处理，所有将来调用 `await()` 的线程不会被暂停。从关闭到开启的变换发生于对 `countDown()` 调用指定次数时。

### Exchanger

Exchanger 是个实用程序，能让两个线程会合并交换一些值。第一个调用 `exchange()` method 的线程会暂停，直到有第二个线程调用了相同的 method。当这个情况发生时，由第一个线程传递给 `exchange()` method 的自变量，就会变成第二个线程的 method 的返回值，反之亦然。当两个 `exchange()` 调用返回时，这两个线程都能随意继续并行运行。Exchanger 是 generic 类型且使用其类型参数来指定要被交换的值的类型。

### CyclicBarrier

CyclicBarrier 是个实用程序，能让 N 个线程的组互相等待以达到同步化的时刻。线程的数量是在 CyclicBarrier 第一次被创建时所指定的。线程会调用 `await()` method 来暂停，直到最后一个线程调用 `await()`，所有的线程在那个时刻都会再次继续。与 CountDownLatch 不同的是，CyclicBarrier 会重设其计数值，并且可以立即被再次使用。CyclicBarrier 在并行算法中很有用，并行算法中的计算会被分为许多部分，每个部分都会由独立的线程处理。在这样的算法中，线程通常必须会合，以让他们的部分解答能合并为完整的解答。为了使这样的操作较为容易，CyclicBarrier 构造函数允许你指定最后一个线程所要执行的 Runnable 对象，它会在其他线程被唤醒并重新运行之前先调用 `await()`。此 Runnable 可以提供从各线程的计算组合出解答所需的协调，或指定新的计算给各个线程。

## 线程中断

在说明 `sleep()`、`join()` 和 `wait()` method 的范例中，你或许注意到，对这些 method 的调用都是被封装在捕获 `InterruptedException` 的 try 语句中。这是必要的，因为 `interrupt()` method 能让一个线程中断另一个线程的运行。中断的结果取决于你处理 `InterruptedException` 的方法，一般对于被中断的线程较好的响应是停止运行。另一方面，如果你只是捕获并忽略 `InterruptedException`，中断就仅只是停止线程，而不是暂停。



如果 `interrupt()` method 是在未被暂停的线程上被调用, 线程就会继续运行, 但它的“中断状态”会被设定, 以指出已有请求被中断。线程可以通过调用静态的 `Thread.interrupted()` method 来测试它自己的中断状态, 如果线程已被中断就会返回 `true`, 但副作用就是会清除中断状态。线程可以用 `isInterrupted()` 这个 instance method 来测试另一个线程上的中断状态, 此 method 会查询状态, 但不会加以清除。

如果线程在中断状态被设定时调用 `sleep()`、`join()` 或 `wait()`, 那么它不会被冻结, 而是立即抛出 `InterruptedException` (中断状态会因为抛出异常所造成的副作用而被清除)。同样地, 如果调用 `interrupt()` method 的线程已在对 `sleep()`、`join()` 或 `wait()` 的调用中被冻结, 那个线程就会因为抛出 `InterruptedException` 而不再被冻结。

线程冻结的最常见情况之一, 就是在进行输入/输出时, 线程通常必须暂停并等待来自文件系统或网络的数据 (用来执行 I/O 操作的 `java.io`、`java.net` 以及 `java.nio` API 会在本章稍后讨论)。很不幸的是, `interrupt()` method 不会唤醒冻结在 `java.io` 包的 I/O method 中的线程。这是 `java.io` 的缺点之一, 已通过 `java.nio` 的 New I/O API 修正。如果线程在执行任一实现了 `java.nio.channels.InterruptibleChannel` 信道上的 I/O 操作时被中断, 信道就会关闭, 线程的中断状态会被设定, 而且线程会通过抛出 `java.nio.channels.ClosedByInterruptException` 被唤醒。如果线程在中断状态已被设定时试图调用阻塞式 I/O method, 就会发生相同的事情。同样地, 如果线程在被冻结在 `java.nio.channels.Selector` 的 `select()` method 中时被中断 (或是在其中断状态已被设定时调用 `select()`), `select()` 就会停止冻结 (或永不启动) 并立即返回。在这样的情况下, 不会有异常被抛出; 被中断的线程会醒来, 而 `select()` 调用会返回。

## 阻塞式队列

就如本章前面的“Queue 与 BlockingQueue 接口”一节中所提到的, 队列是个 collection, 其中的元素会在“尾端”被插入并在“前端”被移除。Queue interface 和各种实现被加入 `java.util` 作为 Java 5.0 的一部分。`java.util.concurrent` 扩展了 Queue interface; BlockingQueue 定义了 `put()` 和 `take()` method, 能让你增加或移除队列的元素, 它在有必要时会被冻结, 直到队列有空间或有元素可被移除为止。阻塞式队列在多线程程序设计中常被使用: 一个线程产生了一些对象并把它们放在队列中以供另一个队列消耗——将那些对象从队列移除。

`java.util.concurrent` 提供了五个 BlockingQueue 的实现:

### ArrayBlockingQueue

此实现是以数组为基础,而且就和所有的数组一样,具有在创建时就已建立的固定能力。以降低总处理能力为代价,此队列可在“尚可”模式中操作,线程会因`put()`或`take()`而被冻结,元素的取用是依它们到达的顺序。

### LinkedBlockingQueue

此实现是以链接列表数据结构为基础。它可以有指定的最大大小,但按照默认,它实际上是无限的。

### PriorityBlockingQueue

此无限队列没有实现 FIFO (先进先出) 顺序,而是以指定 `Comparator` 对象为基础来排序其元素,或者如果它们是 `Comparable` 对象而且没有被指定 `Comparator` 时,就是以它们的自然顺序为基础。由 `take()` 返回的元素是依据 `Comparator` 或 `Comparable` 排序的最小元素。关于非阻塞的版本,请参阅 `java.util.PriorityQueue`。

### DelayQueue

`DelayQueue` 就像 `PriorityBlockingQueue`,但元素实现了 `Delayed interface`。`Delayed` 是 `Comparable` 而且依元素被延迟的时间来排序,但 `DelayQueue` 只是个对元素排序的无限队列。它也限制了 `take()` 和相关的 `method`, 以让元素要到延迟时间已过时才能从队列中被移除。

### SynchronousQueue

此类实现了具有能力为零的 `BlockingQueue` 的退化状况。对 `put()` 的调用会冻结,直到有其他的线程调用了 `take()`, 并且此时对 `take()` 的调用会冻结,直到有其他线程调用 `put()`。

## atomic 变量

`java.util.concurrent.atomic` 包包含了实用程序类,能在不锁定的情况下在字段上进行原子操作 (atomic operation)。原子操作是不可分割的操作: 其他的线程无法看到在原子操作中的 `atomic` 变量。这些实用程序类定义了 `get()` 和 `set()` 访问 `method`, 它们具有 `volatile` 字段的特性,但也定义了如“比较与设定 (compare-and-set)”和“取得与递增 (get-and-increment)”这样行为上的一个原子的复合操作。以下的程序代码示范了 `AtomicInteger` 的用法, 并传统 `synchronized method` 的用法作了对比:

```
// count1()、count2() 和 count3() 都是具有线程安全性的。两个
// 线程可以同时调用这些 method, 而且它们绝不会看到相同的返回值
public class Counters {
    // 这个计数器使用了 synchronized method 和锁定
    int count1 = 0;
    public synchronized int count1() { return count1++; }
```

```

// 这个计数器使用了在 AtomicInteger 上的单元递增
AtomicInteger count2 = new AtomicInteger(0);
public int count2() { return count2.getAndIncrement(); }

// 这个乐观的计数器使用了 compareAndSet()
AtomicInteger count3 = new AtomicInteger(0);
public int count3() {
    // 使用 get() 来取得计数器值, 并使用 compareAndSet() 加以设定
    // 如果 compareAndSet() 返回 false, 就再试一下, 直到我们在没有
    // 阻塞的情况下处理完循环
    int result;
    do {
        result = count3.get();
    } while(!count3.compareAndSet(result, result+1));
    return result;
}
}

```

## 文件与目录

java.io.File 类表示了文件或目录, 并定义了一些重要的 method 来操作文件和目录。但请注意, 这些 method 没有允许你读取文件的内容, 那是 java.io.FileInputStream 的工作——它只是 Java 中用到的许多种 I/O 流中的一种, 会在接下来的章节中讨论。这里有一些你可以用 File 做的事:

```

import java.io.*;
import java.util.*;

// 取得用户根目录的名称并以 File 来表示
File homedir = new File(System.getProperty("user.home"));
// 创建一个 File 对象来表示那个目录中的文件
File f = new File(homedir, ".configfile");

// 判断文件的大小以及最后修改时间
long filelength = f.length();
Date lastModified = new java.util.Date(f.lastModified());

// 如果文件存在又不是个目录, 而且具有可读性,
// 就把它移到新建的目录中
if (f.exists() && f.isFile() && f.canRead()) {
    File configdir = new File(homedir, ".configdir"); // 检查配置文件
    configdir.mkdir(); // 新的配置目录
    f.renameTo(new File(configdir, ".config")); // 创建那个目录
} // 将文件移入

// 列出根目录中的所有文件
String[] allfiles = homedir.list();

// 列出具有 ".java" 后缀的所有文件
String[] sourcecode = homedir.list(new FilenameFilter() {
    public boolean accept(File d, String name) { return name.endsWith(".java"); }
});

```



```

The File class gained some important additional functionality as of Java 1.2:
// 列出所有文件系统根目录; 在 Windows 上, 这会给我们针对所有驱动器
// 盘符的 File 对象 (适用于 Java 1.2 及之后的版本)
File[] rootdirs = File.listRoots();

// 创建一个锁定文件, 然后删除 (适用于 Java 1.2 及之后的版本)
File lock = new File(configdir, ".lock");
if (lock.createNewFile()) {
    // 我们成功地创建了这个文件, 现在安排在退出时删除它
    lock.deleteOnExit();

    // 现在执行应用程序安全性, 让其他人不能
    // 同时执行它
    ...
}
else {
    // 我们没有创建文件; 其他人占有了锁
    System.err.println("Can't create lock file; exiting.");
    System.exit(1);
}

// 创建临时文件以供处理过程中使用 (适用于 Java 1.2 及之后的版本)
File temp = File.createTempFile("app", ".tmp"); // 文件名的前缀和后缀
// 使用 temp 文件做一些事
...
// 在完成时删除它
temp.delete();

```

## RandomAccessFile

java.io 包也定义了 RandomAccessFile 类, 能让你从文件中的任意位置读取二进制数据。这在指定的情况中很有用, 但大部分的应用程序会使用下一章节所描述的流类来循序读取文件。这有个使用 RandomAccessFile 的简短范例:

```

// 打开文件以供读取 / 写入 ("rw") 访问
File datafile = new File(configdir, "datafile");
RandomAccessFile f = new RandomAccessFile(datafile, "rw");
f.seek(100); // 移至文件的第 100 字节处
byte[] data = new byte[100]; // 创建缓冲区来保存数据
f.read(data); // 从文件读取 100 个字节
int i = f.readInt(); // 从文件读取 4 字节的整数
f.seek(100); // 退回第 100 字节的地方
f.writeInt(i); // 先写入整数
f.write(data); // 然后写入 100 个字节
f.close(); // 在完成时关闭文件

```

## 使用 java.io 输入 / 输出

java.io 包定义了大量的类供读取与写入流 (或循序) 数据。InputStream 和 OutputStream 类是用于读取与写入字节流, 而 Reader 和 Writer 类是用于读取与写

入字符串流。流可以被嵌套，这代表你可以从读取并处理来自底层 Reader 流的 FilterReader 对象读取字符。这个底层的 Reader 流可以从 InputStream 读取字节并把它们转换为字符。

## 读取控制台输入

你可以用流执行一些常见的操作，其中一项是读取用户在控制台的输入：

```
import java.io.*;

BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
System.out.print("What is your name: ");
String name = null;
try {
    name = console.readLine();
}
catch (IOException e) { name = "<" + e + ">"; } // 这绝不该发生
System.out.println("Hello " + name);
```

## 从文本文件读取多行

从文件读取多行文本是个类似的操作。以下程序代码会读取整个文本文件并在读到结尾时结束：

```
String filename = System.getProperty("user.home") + File.separator + ".cshrc";
try {
    BufferedReader in = new BufferedReader(new FileReader(filename));
    String line;
    while((line = in.readLine()) != null) { // 读取一行，检查是否为文件结尾
        System.out.println(line);         // 列出此行
    }
    in.close(); // 当处理完成后一定要关闭流
}
catch (IOException e) {
    // 在这处理 FileNotFoundException 等异常
}
```

## 将文本写入文件

在本书各处，你都会看到 System.out.println() method 被用来将文字显示在控制台上。System.out 仅只是引用输出流，你可以用类似的技术将文字输出在任何的输出流上。以下程序代码显示如何将文字输出至文件：

```
try {
    File f = new File(homedir, ".config");
    PrintWriter out = new PrintWriter(new FileWriter(f));
```

```
        out.println("## Automatically generated config file. DO NOT EDIT!");
        out.close(); // 已完成写入
    }
    catch (IOException e) { /* 处理异常 */ }
```

## 读取二进制文件

但是，不是所有的文件都包含文本。以下程序代码将文件视为字节流，并将字节读入大数组中：

```
try {
    File f; // 要读取的文件，已在其他地方初始化
    int filesize = (int) f.length(); // 计算文件大小
    byte[] data = new byte[filesize]; // 创建足够大的数组
    // 创建一个流来读取文件
    DataInputStream in = new DataInputStream(new FileInputStream(f));
    in.readFully(data); // 将文件内容读入数组
    in.close();
}
catch (IOException e) { /* 处理异常 */ }
```

## 压缩数据

其他各种Java平台的包定义了专门的流类，来以一些有效的方式操作流数据。以下程序代码说明如何使用来自 java.util.zip 的流类来计算数据的校验码 (checksum)，然后在将数据写入文件时加以压缩：

```
import java.io.*;
import java.util.zip.*;

try {
    File f; // 要写入的文件，已在其他地方初始化
    byte[] data; // 要写入的数据，已在其他地方初始化
    Checksum check = new Adler32(); // 用来计算简单校验码的对象

    // 创建一个流，将字节写入文件 f
    FileOutputStream fos = new FileOutputStream(f);
    // 创建一个流，将字节压缩并写入 fos
    GZIPOutputStream gzos = new GZIPOutputStream(fos);
    // 创建一个流，计算写入 gzos 的字节的校验码
    CheckedOutputStream cos = new CheckedOutputStream(gzos, check);

    cos.write(data); // 现在将数据写入嵌套流
    cos.close(); // 关闭流的嵌套链
    long sum = check.getValue(); // 取得计算好的校验码
}
catch (IOException e) { /* 处理异常 */ }
```



## 读取 ZIP 文件

java.util.zip包也包含了ZipFile类,能让你随机访问ZIP文件的入口并让你通过流读取那些入口:

```
import java.io.*;
import java.util.zip.*;
String filename; // 要读取的文件,已在其他地方初始化
String entryname; // 从ZIP文件读出的入口,已在其他地方初始化
ZipFile zipfile = new ZipFile(filename); // 打开ZIP文件
ZipEntry entry = zipfile.getEntry(entryname); // 取得一个入口
InputStream in = zipfile.getInputStream(entry); // 用来读取入口的流
BufferedInputStream bis = new BufferedInputStream(in); // 改善效率
// 现在从bis读取字节.....
// 输出ZIP文件的内容
for(java.util.Enumeration e = zipfile.entries(); e.hasMoreElements();) {
    ZipEntry zipentry = (ZipEntry) e.nextElement();
    System.out.println(zipentry.getName());
}
```

## 计算消息摘要

如果你必须计算具有密码强度的校验码(也就是所谓的消息摘要(message digest)),可以使用java.security包中的一个流类。例如:

```
import java.io.*;
import java.security.*;
import java.util.*;

File f; // 用来读取及计算摘要的文件,已在其他地方初始化
List text = new ArrayList(); // 我们在这存储成行的文本

// 取得可以计算SHA消息摘要的对象
MessageDigest digester = MessageDigest.getInstance("SHA");
// 用来从文件f读取字节的流
FileInputStream fis = new FileInputStream(f);
// 此流会从fis读取字节,并计算SHA消息摘要
DigestInputStream dis = new DigestInputStream(fis, digester);
// 此流会从dis读取字节,并将它们转换为字符
InputStreamReader isr = new InputStreamReader(dis);
// 此流一次可以读取一行
BufferedReader br = new BufferedReader(isr);
// 现在从流读取文本行
for(String line; (line = br.readLine()) != null; text.add(line)) ;
// 关闭流
br.close();
// 取得消息摘要
byte[] digest = digester.digest();
```

## 将数据存入数组与从数组取出

到目前为止，我们使用了各种流类来操作流数据，但数据本身最终是来自文件或会被写入控制台。java.io包定义了其他的流类，可以从字节数组或文本字符串读取与写入数据：

```
import java.io.*;

// 设定一个流，它使用字节数组作为目的地
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream out = new DataOutputStream(baos);
out.writeUTF("hello");           // 将一些字符串数据当作字节写出
out.writeDouble(Math.PI);        // 将浮点值当作字节写出
byte[] data = baos.toByteArray(); // 取得我们写出的字节数组
out.close();                     // 关闭流

// 设定一个流从字符串读取字符
Reader in = new StringReader("Now is the time!");
// 读取字符直到达到终点
int c;
while((c = in.read()) != -1) System.out.print((char) c);
```

以此方式操作的其他类包括了ByteArrayInputStream、StringWriter、CharArrayReader以及CharArrayWriter。

## 使用管道进行线程通信

PipedInputStream和PipedOutputStream以及它们的基于字符的另两个类——PipedReader和PipedWriter，是由java.io所定义的另一组有趣的流。这些流会被两个想要进行通信的线程成对使用。一个线程会写入字节到PipedOutputStream或写入字符到PipedWriter，而另一个线程会从相应的PipedInputStream或PipedReader读取字节或字符：

```
// 一对连通的管道 I/O 流形成了一个管道。一个线程会写入字节到
// PipedOutputStream，而另一个线程会从相应的 PipedInputStream
// 读取它们，或针对字符使用 PipedWriter/PipedReader
final PipedOutputStream writeEndOfPipe = new PipedOutputStream();
final PipedInputStream readEndOfPipe = new
PipedInputStream(writeEndOfPipe);

// 此线程会从管道读取字节并抛弃它们
Thread devnull = new Thread(new Runnable() {
    public void run() {
        try { while(readEndOfPipe.read() != -1); }
        catch (IOException e) {} // 忽略它
    }
});
devnull.start();
```

## 使用 java.net 进行网络连接

java.net 包定义了一些类，它们让网络应用程序的编写变得超乎想象的容易。各种范例如下：

### 使用 URL 类进行网络连接

最易于使用的网络连接类就是 URL，它表示全球资源定位器 (uniform resource locator)。不同的 Java 实现可能支持不同的 URL 协议，但至少你可以利用其对 http://、ftp:// 以及 file:// 的支持。在 Java 1.4 中，安全 HTTP (secure HTTP) 也用 https:// 协议来支持。这有一些使用 URL 类的方法：

```
import java.net.*;
import java.io.*;

// 创建一些 URL 对象
URL url=null, url2=null, url3=null;
try {
    url = new URL("http://www.oreilly.com");           // 绝对 URL
    url2 = new URL(url, "catalog/books/javanut4/");    // 相对 URL
    url3 = new URL("http:", "www.oreilly.com", "index.html");
} catch (MalformedURLException e) { /* 忽略此异常 */ }

// 从输入流读取 URL 的内容
InputStream in = url.openStream();

// 为能对读取过程做更多的控制，取得 URLConnection 对象
URLConnection conn = url.openConnection();

// 现在取得一些关于 URL 的信息
String type = conn.getContentType();

String encoding = conn.getContentEncoding();
java.util.Date lastModified = new java.util.Date(conn.getLastModified());
int len = conn.getContentLength();

// 如果有需要，就使用流读取 URL 的内容
InputStream in = conn.getInputStream();
```

### 运用 Socket

有时你需要对网络应用程序做比 URL 类所能做到的程度还高的控制。在这样的情况下，你可以使用 Socket 直接与服务器通信。例如：

```
import java.net.*;
import java.io.*;

// 这是个简单的客户端程序，它会联机至 web server，请求一个
// 文件并从服务器读取文件
```



```
String hostname = "java.oreilly.com"; // 要联机的服务器
int port = 80; // HTTP 的标准端口
String filename = "/index.html"; // 要从服务器读取的文件
Socket s = new Socket(hostname, port); // 联机至服务器

// 取得可用来与服务器通信的 I/O 流
InputStream sin = s.getInputStream();
BufferedReader fromServer = new BufferedReader(new InputStreamReader(sin));
OutputStream sout = s.getOutputStream();
PrintWriter toServer = new PrintWriter(new OutputStreamWriter(sout));

// 使用 HTTP 协议请求来自服务器的文件
toServer.print("GET " + filename + " HTTP/1.0\r\n\r\n");
toServer.flush();

// 现在读取服务器的响应, 假设它是个文本文件并把它输出
for(String l = null; (l = fromServer.readLine()) != null; )
    System.out.println(l);

// 在完成时把一切关闭
toServer.close();
fromServer.close();
s.close();
```

## 使用 SSL 处理 Secure Socket

在 Java 1.4 中, Java Secure Socket Extension (或 JSSE) 被加到包 `javax.net` 和 `javax.net.ssl` 中的核心 Java 平台。此 API 加密了在 socket 上使用 SSL (Secure Sockets Layer, 也就是所谓的 TLS) 协议的网络通信。SSL 被广泛用于 Internet 上: 它是使用了 `https://` 协议的安全 web 通信的基础。在 Java 1.4 和之后的版本中, 你可以用 `https://` 配合如前所示的 URL 类, 来安全地下载来自支持 SSL 的 web 服务器的文件。

就和所有的 Java 安全 API 一样, JSSE 具高度可配置性, 并对所有在 SSL socket 上设定与通信的细节给予低级的控制。`javax.net` 和 `javax.net.ssl` 包相当复杂, 但实际上, 你只需要一些类来安全地与服务器通信。以下程序是先前的程序代码的变体, 它使用了 HTTPS 来代替 HTTP, 以安全地传输请求 URL 的内容:

```
import java.io.*;
import java.net.*;
import javax.net.ssl.*;
import java.security.cert.*;

/**
 * 使用 HTTPS 从 web 服务器取得文件。用法:
 * java HttpsDownload <hostname> <filename>
 */
public class HttpsDownload {
    public static void main(String[] args) throws IOException {
        // 取得用来建立 SSL socket 的 SocketFactory 对象
```

```

SSLSocketFactory factory =
    (SSLSocketFactory) SSLSocketFactory.getDefault();

// 使用 factory 来建立 secure socket, 联机至
// 指定 web 服务器的 HTTPS 端口
SSLSocket sslsock=(SSLSocket)factory.createSocket(args[0], // 主机名称
                                                    443); // HTTPS 端口

// 取得由 web 服务器所提供的证书
SSLSession session = sslsock.getSession();
X509Certificate cert;
try { cert = (X509Certificate)session.getPeerCertificates()[0]; }
catch(SSLPeerUnverifiedException e) { // 如果没有证书或证书无效
    System.err.println(session.getPeerHost() +
                        " did not present a valid certificate.");
    return;
}

// 显示关于认证的细节
System.out.println(session.getPeerHost() +
                    " has presented a certificate belonging to:");
System.out.println("\t[" + cert.getSubjectDN().getName() + "]);
System.out.println("The certificate bears the valid signature of:");
System.out.println("\t[" + cert.getIssuerDN().getName() + "]);

// 如果用户不信任此认证就放弃
System.out.print("Do you trust this certificate (y/n)? ");
System.out.flush();
BufferedReader console =
    new BufferedReader(new InputStreamReader(System.in));
if (Character.toLowerCase(console.readLine().charAt(0)) != 'y') return;

// 现在使用 secure socket 就像使用正常的 socket 一样
// 首先, 在 SSL socket 上送出正常的 HTTP 请求
PrintWriter out = new PrintWriter(sslsock.getOutputStream());
out.print("GET " + args[1] + " HTTP/1.0\r\n\r\n");
out.flush();

// 接着, 读取服务器的响应并将它输送至控制台
BufferedReader in =
    new BufferedReader(new InputStreamReader(sslsock.getInputStream()));
String line;
while((line = in.readLine()) != null) System.out.println(line);

// 最后, 关闭 socket
sslsock.close();
}
}

```

## 服务器

客户端应用程序使用 Socket 与服务器通信, 服务器也做了相同的事: 它使用 Socket 对象与各个客户端通信。但是, 服务器有额外的工作, 它必须能识别并接收客户端的联

机请求。这是用 ServerSocket 类实现的。以下程序代码展示了使用 ServerSocket 的方法。此程序代码实现了简单的 HTTP 服务器，它会通过回送（或镜像）HTTP 请求的全部内容来响应所有的请求。在对 HTTP 客户端调试时，像这样的虚拟服务器是很有用的：

```
import java.io.*;
import java.net.*;

public class HttpMirror {
    public static void main(String[] args) {
        try {
            int port = Integer.parseInt(args[0]);           // 要监听的端口
            ServerSocket ss = new ServerSocket(port);        // 建立 socket 来监听
            for(;;) {                                        // 无限循环
                Socket client = ss.accept();                 // 等待联机
                ClientThread t = new ClientThread(client);   // 用一个线程来处理
                t.start();                                   // 让线程开始运行
            }                                                // 再次循环
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.err.println("Usage: java HttpMirror <port>;");
        }
    }

    static class ClientThread extends Thread {
        Socket client;
        ClientThread(Socket client) { this.client = client; }
        public void run() {
            try {
                // 让流与客户端通信
                BufferedReader in =
                    new BufferedReader(new InputStreamReader(client.getInputStream()));
                PrintWriter out =
                    new PrintWriter(new OutputStreamWriter(client.getOutputStream()));

                // 发送 HTTP 响应报头至客户端
                out.print("HTTP/1.0 200\r\nContent-Type: text/plain\r\n\r\n");

                // 读取来自客户端的 HTTP 请求并立刻响应
                // 当我们从客户端读到用来发送请求及其发送结束
                // 的空白行时就停止

                String line;
                while((line = in.readLine()) != null) {
                    if (line.length() == 0) break;
                    out.println(line);
                }

                out.close();
                in.close();
                client.close();
            }
        }
    }
}
```



```

        catch (IOException e) { /* 忽略异常 */ }
    }
}

```

此服务器程序代码可以用 JSSE 来修改以支持 SSL 联机。但是，让服务器具有安全性是比让客户端具有安全性更复杂的，因为服务器必须具有可以给客户端的证书。因此，服务器端的 JSSE 不在此处示范。

## 数据报 (datagram)

URL 和 Socket 都是在基于流的网络联机上执行网络连接的。但是，要跨网络设定与维护流，则需要在网络层上做一些事。有时你需要以低层次的方法来加速跨网络的数据包传输速度，但你不必在意流的维护。此外，如果你不需要保证数据包的到达或数据包以和发送相同的顺序到达，就会对 DatagramSocket 和 DatagramPacket 类感兴趣：

```

import java.net.*;

// 使用 datagram 送出信息到另一台计算机
try {
    String hostname = "host.example.com"; // 数据所要送达的计算机
    InetAddress address = // 将 DNS 主机名称转换为
        InetAddress.getByName(hostname); // 低层次的 IP 地址
    int port = 1234; // 所要联机的端口
    String message = "The eagle has landed."; // 要送出的信息
    byte[] data = message.getBytes(); // 将字符串转换为字节
    DatagramSocket s = new DatagramSocket(); // 要送出信息的 socket
    DatagramPacket p = // 创建包以传送
        new DatagramPacket(data, data.length, address, port);
    s.send(p); // 现在进行传送!
    s.close(); // 在完成时一定要关闭 socket
}
catch (UnknownHostException e) {} // 由 InetAddress.getByName() 抛出
catch (SocketException e) {} // 由 new DatagramSocket() 抛出
catch (java.io.IOException e) {} // 由 DatagramSocket.send() 抛出

// 这里是其他计算机收到 datagram 的方式
try {
    byte[] buffer = new byte[4096]; // 用来保存数据的缓冲区

    DatagramSocket s = new DatagramSocket(1234); // 用来接收的 socket
    DatagramPacket p =
        new DatagramPacket(buffer, buffer.length); // 要被接收的包
    s.receive(p); // 等待包到达
    String msg = // 将包从字节转
        new String(buffer, 0, p.getLength()); // 换为字符串
    s.close(); // 一定要关闭 socket
}
catch (SocketException e) {} // 由 new DatagramSocket() 抛出
catch (java.io.IOException e) {} // 由 DatagramSocket.receive() 抛出

```

## 测试主机的可达性 (reachability)

在 Java 5.0 中, `InetAddress` 类有个 `isReachable()` method, 它会尝试判定主机是否可达。以下程序代码在 Unix *ping* 实用程序的原生 Java 实现中使用:

```
import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;

public class Ping {
    public static void main(String[] args) throws IOException {
        try {
            String hostname = args[0];
            int timeout = (args.length > 1)?Integer.parseInt(args[1]):2000;
            InetAddress[] addresses = InetAddress.getAllByName(hostname);
            for(InetAddress address : addresses) {
                if (address.isReachable(timeout))
                    System.out.printf("%s is reachable\n", address);
                else
                    System.out.printf("%s could not be contacted\n", address);
            }
        }
        catch (UnknownHostException e) {
            System.out.printf("Unknown host: %s\n", args[0]);
        }
        catch(IOException e) { System.out.printf("Network error: %s\n", e); }
        catch (Exception e) {
            // ArrayIndexOutOfBoundsException 或 NumberFormatException
            System.out.println("Usage: java Ping <hostname> [timeout in ms]");
        }
    }
}
```

## 使用 java.nio 进行 I/O 与网络连接

Java 1.4 针对高性能、非阻塞式 (nonblocking) I/O 和网络连接引入了全新的 API。此 API 主要是由三个新包组成: `java.nio` 定义了 `Buffer` 类, 用来存储一连串的字节或其他基本值; `java.nio.channels` 定义了信道, 数据可通过它在缓冲区和数据来源地或目的端 (data source or sink, 例如文件或网络 socket) 之间传输, 此包也包含了用于非阻塞式 I/O 的重要类; 最后, `java.nio.charset` 包包含了有效地将字节缓冲区转换为字符缓冲区的类。以下章节包含了使用这三种包的范例以及使用 New I/O API 进行特定 I/O 工作的范例。

### 基本 Buffer 操作

`java.nio` 包包含了抽象的 `Buffer` 类, 它定义了缓冲区上的一般操作。此包也定义了类型特有的子类, 例如 `ByteBuffer`、`CharBuffer` 和 `IntBuffer`。以下程序代码说

明在 `ByteBuffer` 上的典型的序列缓冲区操作。其他类型特有的缓冲区类也有类似的 `method`。

```
import java.nio.*;

// Buffer 不具有公用构造函数，它们是被分配的
ByteBuffer b = ByteBuffer.allocate(4096); // 创建 4096 字节的缓冲区
// 或是这样做来试着从低层次的 OS 取得有效的缓冲区
ByteBuffer buf2 = ByteBuffer.allocateDirect(65536);
// 这是另一个取得缓冲区的方法：通过“封装”一个数组
byte[] data; // 假设数组已创建且在其他地方初始化
ByteBuffer buf3 = ByteBuffer.wrap(data); // 创建使用数组的缓冲区
// 创建“查看缓冲区”以把字节当成其他类型来查看也是有可能的
buf3.order(ByteOrder.BIG_ENDIAN); // 指定缓冲区的字节顺序
IntBuffer ib = buf3.asIntBuffer(); // 把那些字节当作整数来查看

// 现在存储一些数据到缓冲区
b.put(data); // 把数组的字节复制到缓冲区的当前位置
b.put((byte)42); // 把另一个字节存储在新的当前位置
b.put(0, (byte)9); // 改写缓冲区中的第一个字节，不改变位置
b.order(ByteOrder.BIG_ENDIAN); // 设定缓冲区的字节顺序
b.putChar('x'); // 把 2 个字节的 Unicode 字符存储在缓冲区
b.putInt(0xcafebabe); // 把 4 个字节的 int 存储到缓冲区

// 这些是查询关于缓冲区基本数值的 method
int capacity = b.capacity(); // 此缓冲区可以保存多少字节？(4,096)
int position = b.position(); // 下个要被写入或读取的字节在哪里？
// 缓冲区的界限指定了缓冲区有多少字节可以被使用
// 在写入到缓冲区时，这应该就是容量；从缓冲区读取数据
// 时，这应该就是先前写入的字节数量
int limit = b.limit(); // 应该要使用多少？
int remaining = b.remaining(); // 还剩下多少？返回界限位置
boolean more = b.hasRemaining(); // 测试缓冲区中是否还有空间

// 位置 (position) 和界限 (limit) 也可以用具有相同名称的 method 设定
// 假设你要读取已写入缓冲区的字节
b.limit(b.position()); // 将界限设定为当前位置
b.position(0); // 将界限设定为 0，从开头开始读取

// 你通常不会用前面的两个调用，而是会用方便的 method
b.flip(); // 将界限设定为位置，位置设定为 0；准备读取
b.rewind(); // 将位置设定为 0，不改变界限；准备读取
b.clear(); // 将位置设定为 0，界限设定为容量；准备写入

// 假设你已调用 flip()，就可以开始从缓冲区读取字节
buf2.put(b); // 从 b 读取所有字节并把它们放入 buf2
b.rewind(); // 将 b 倒回以便从开头再次读取
byte b0 = b.get(); // 读取第一个字节；递增缓冲区位置
byte b1 = b.get(); // 读取第二个字节；递增缓冲区位置
byte[] fourbytes = new byte[4];
b.get(fourbytes); // 读取接下来的 4 个字节并将缓冲区位置加 4
byte b9 = b.get(9); // 读取第 10 个字节，但不改变当前位置
int i = b.getInt(); // 把接下来的 4 个字节当成整数来读取；位置加 4
```



```
// 丢弃已读取的字节，将剩余的移转至缓冲区的开  
// 头；将位置设定为界限，将界限设定为容量，准备  
// 好缓冲区以供写入更多字节  
b.compact();
```

你或许注意到，许多缓冲区 method 会返回它们所操作的对象。这让 method 调用在程序代码中可以被“链接”，如下：

```
ByteBuffer bb=ByteBuffer.allocate(32).order(ByteOrder.BIG_ENDIAN).putInt(1234);
```

在 java.nio 及其子包中的许多 method 会返回当前对象，使得这种 method 链接成为可能。请注意，使用这种链接是个风格上的选择（我在本章中避开了），在效率上并没有重大的影响。

ByteBuffer 是最重要的缓冲区类。然而，另一个常用的类是 CharBuffer。CharBuffer 对象可以通过封装字符串来创建，也可以被转换为字符串。CharBuffer 实现了新的 java.lang.CharSequence interface，这代表在特定的应用程序中，它可以像 String 或 StringBuffer 一样被使用（亦即用于正则表达式模式匹配）。

```
// 从字符串创建只读 CharBuffer  
CharBuffer cb = CharBuffer.wrap("This string is the data for the  
CharBuffer");  
String s = cb.toString(); // 用 toString() method 或依据 toString() 的  
System.out.println(cb); // 隐式调用来转换为字符串  
char c = cb.charAt(0); // 使用 CharSequence method 来取得字符  
char d = cb.get(1); // 或使用 CharBuffer 绝对位置来读取  
// 相对位置读取字符并递增当前位置  
// 请注意，当 CharBuffer 被转换为 String 或被当成 CharSequence 使用  
// 时，只有当前位置到界限的字符会被使用  
char e = cb.get();
```

ByteBuffer 中的字节通常会被转换为 CharBuffer 中的字符，反之亦然。当我们考虑到 java.nio.charset 包时，就会看到要如何操作。

## 基本信道操作

缓冲区本身并没有什么用，把字节存入缓冲区只是要把它们再次读出，其他没什么内容。但是，缓冲区通常会配合信道一起使用：你的程序把字节存入缓冲区，然后把缓冲区传递给信道，信道把字节从缓冲区读出并将它们写入文件、网络 socket 或其他目的地。或者反过来看，你的程序把缓冲区传给信道，信道从文件、socket 或其他来源地读取字节并把那些字节存储在缓冲区中，然后这些字节就能被你的程序获取。java.nio.channels 包定义了代表文件、socket、datagram 以及管道等信道类（我们会在本章的稍后看到这些具体类的范例）。但是，以下程序代码是以由 java.nio.channels 所定义的各种信道 interface 的功能为基础，应该能配合任一个 Channel 对象运行：

```

Channel c; // 实现了Channel interface的对象;已在其他地方初始化
if (c.isOpen()) c.close(); // 这是唯一由Channel定义的method

// read()和write() method是由ReadableByteChannel
// 和WritableByteChannel interface所定义
ReadableByteChannel source; // 已在其他地方初始化
WritableByteChannel destination; // 已在其他地方初始化
ByteBuffer buffer = ByteBuffer.allocateDirect(16384); // 低层次的16 KB缓冲区

// 这是用于从源信道读取字节并将它们写入目的信道
// 的基本循环,它会执行到不再有字节可从来源地读取以
// 及不再有已在缓冲区的字节要写入目的地为止
while(source.read(buffer) != -1 || buffer.position() > 0) {
    // 滚动缓冲区:将界限设定为当前位置,当当前位置设定为0。这会
    // 让缓冲区准备好供读取(这是由信道 * 写入 * 操作完成)
    buffer.flip();
    // 将缓冲区中的一些或全部字节写入目的地
    destination.write(buffer);
    // 丢弃已被写入的字节,将剩余的复制到缓冲区的开头。将当前
    // 位置设定为界限,将界限设定为容量,将缓冲区准备好以供写入
    // (这是由信道 * 读取 * 操作完成)
    buffer.compact();
}

// 不要忘记关闭信道
source.close();
destination.close();

```

除了先前的程序代码中介绍的 `ReadableByteChannel` 和 `WritableByteChannel` interface 之外, `java.nio.channels` 还定义了几个其他的信道 interface。 `ByteChannel` 只是扩展具有可读性和具有可写性的 interface, 但没有增加任何的新 method。 它是有用的信道简略表达方式, 同时支持读取与写入。 `GatheringByteChannel` 是 `WritableByteChannel` 的扩展, 它定义了 `write()` method, 会从多个缓冲区收集数据并将数据写至多个缓冲区。 同样地, `ScatteringByteChannel` 是 `ReadableByteChannel` 的扩展, 它定义了 `read()` method, 会从信道读取字节并把它们传播或分布到多个缓冲区。 当处理使用固定大小报头的网络协议, 而且你想把报头与其他要被传输的数据分开存储在缓冲区时, 收集和传播的 `write()` 与 `read()` method 会很有用。

有个会令人混淆、必须意识到的一点, 就是信道读取操作牵涉到将字节写入 (或放入) 缓冲区, 而信道写入操作牵涉到从缓冲区读取 (或取得) 字节。 因此, 当我说 `flip()` method 准备好缓冲区以供读取时, 我是指它准备了缓冲区以供信道 `write()` 操作使用! 对于缓冲区的 `compact()` method, 相反的操作也成立。

## 使用 Charset 编码与译码文字

`java.nio.charset.Charset` 对象代表一个字符集以及那个字符集的编码。 `Charset`

以及与其相结合的类, `CharsetEncoder` 和 `CharsetDecoder`, 定义了一些 `method` 用来将字符组成的字符串编码为字节序列以及将字节序列译码为由字符组成的字符串。由于这些类是 New I/O API 的一部分, 它们使用了 `ByteBuffer` 和 `CharBuffer` 类:

```
// 最简单的情况。使用 Charset 的便捷例程来转换
Charset charset = Charset.forName("ISO-8859-1"); // 取得 Latin-1 Charset
CharBuffer cb = CharBuffer.wrap("Hello World"); // 要编码的字符
// 将字符编码并将字节存储在新分配的 ByteBuffer 中
ByteBuffer bb = charset.encode(cb);
// 将这些字节译码为新分配的 CharBuffer 并把它们输出
System.out.println(charset.decode(bb));
```

请注意此范例中使用的 ISO-8859-1 (又名 Latin-1) 字符集 (`charset`)。此 8 位字符集适用于大部分的西欧语言, 其中包括英文。只使用英文的程序员也可以使用 7 位的 US-ASCII 字符集。`Charset` 类本身并未进行编码和译码, 而先前的便捷例程在内部创建了 `CharsetEncoder` 和 `CharsetDecoder` 类。如果你计划要编码或译码多次, 那么自行创建这些对象会更有效率:

```
Charset charset = Charset.forName("US-ASCII"); // 取得字符集
CharsetEncoder encoder = charset.newEncoder(); // 从中创建编码器
CharBuffer cb = CharBuffer.wrap("Hello World!"); // 取得 CharBuffer
WritableByteChannel destination; // 已在其他地方初始化
destination.write(encoder.encode(cb)); // 编码字符并写入
```

前面的 `CharsetEncoder.encode()` `method` 在每次被调用时, 都必须分配新的 `ByteBuffer`。如果要取得最大的效率, 你可以调用较低层次的 `method` 将编码和译码处理到现有的缓冲区中:

```
ReadableByteChannel source; // 已在其他地方初始化
Charset charset = Charset.forName("ISO-8859-1"); // 取得字符集
CharsetDecoder decoder = charset.newDecoder(); // 从中创建译码器
ByteBuffer bb = ByteBuffer.allocateDirect(2048); // 用来保存字节的缓冲区
CharBuffer cb = CharBuffer.allocate(2048); // 用来保存字符的缓冲区

while(source.read(bb) != -1) { // 从信道读取字节, 直到 EOF 为止
    bb.flip(); // 滚动字节缓冲区以准备解码
    decoder.decode(bb, cb, true); // 将字节译码至字符
    cb.flip(); // 滚动字符缓冲区以准备输出
    System.out.print(cb); // 输出字符
    cb.clear(); // 清除字符缓冲区
    bb.clear(); // 准备字节缓冲区以供下一个信道读取
}
source.close(); // 已使用完信道, 所以将它关闭
System.out.flush(); // 确定所有的输出字符都已出现
```

前面的程序代码依据的事实是 ISO-8859-1 是 8 位编码字符集以及在字符和字节之间有一对一的映射。但是, 对于较复杂的字符集, 例如 Unicode 的 UTF-8 编码或日文所使用的 EUC-JP 字符集, 一个字节不够用, 有些 (或全部) 字符必须要有多个字节。在遇到这



种情况时,无法保证缓冲区中所有的字节可以一次性被编码(缓冲区的结尾可能包含了不完整的字符)。此外,由于一个字符可能被译码为多个字节,所以要知道给定的字符串会被译码为多少字节是有困难的。以下程序代码显示一个循环,你可以用它来以较一般的方式译码字节:

```

ReadableByteChannel source;                // 已在其他地方初始化
Charset charset = Charset.forName("UTF-8"); // Unicode 编码
CharsetDecoder decoder = charset.newDecoder(); // 从中创建译码器
ByteBuffer bb = ByteBuffer.allocateDirect(2048); // 用来保存字节的缓冲区
CharBuffer cb = CharBuffer.allocate(2048); // 用来保存字符的缓冲区

// 告诉译码器忽略坏字节可能产生的错误
decoder.onMalformedInput(CodingErrorAction.IGNORE);
decoder.onUnmappableCharacter(CodingErrorAction.IGNORE);

decoder.reset(); // 如果译码器在之前已被使用过,则将它重设
while(source.read(bb) != -1) { // 从信道读取字节,直到 EOF
    bb.flip(); // 滚动字节缓冲区以准备译码
    decoder.decode(bb, cb, false); // 将字节译码为字符
    cb.flip(); // 滚动字符缓冲区以准备输出
    System.out.print(cb); // 输出字符
    cb.clear(); // 清除字符缓冲区
    bb.compact(); // 丢弃已译码的字节
}
source.close(); // 已使用完信道,将它关闭
// 在这时,缓冲区中可能还有一些字节要译码
bb.flip(); // 准备解码
decoder.decode(bb, cb, true); // 返回 true 以指出这是最后一个调用
decoder.flush(cb); // 输出最后一个字符
cb.flip(); // 滚动字符缓冲区
System.out.print(cb); // 输出最后的字符

```

## 文件处理

`FileChannel` 是个具体的 `Channel` 类,它会执行文件 I/O 并实现了 `ReadableByteChannel` 和 `WritableByteChannel` (虽然它的 `read()` method 只有在底层的文件已打开来供读取时才能运作,而它的 `write()` method 只有在文件已开启来供写入时才能运作)。通过使用 `java.io` 包取得 `FileChannel` 对象来创建 `FileInputStream`、`FileOutputStream` 或 `RandomAccessFile`,然后调用那个对象的 `getChannel()` method (Java 1.4 中增加的)。举例来看,你可以用两个 `FileChannel` 对象来复制文件:

```

String filename = "test"; // 要复制的文件名称
// 创建流来读取源文件并写至副本
FileInputStream fin = new FileInputStream(filename);
FileOutputStream fout = new FileOutputStream(filename + ".copy");
// 使用流来创建相对应的信道对象
FileChannel in = fin.getChannel();
FileChannel out = fout.getChannel();

```

```
// 针对副本配置低层次的 8kB 缓冲区
ByteBuffer buffer = ByteBuffer.allocateDirect(8192);
while(in.read(buffer) != -1 || buffer.position() > 0) {
    buffer.flip();           // 准备从缓冲区读取并写入至文件
    out.write(buffer);       // 写入一些或全部缓冲区内容
    buffer.compact();        // 丢弃所有已写入的字节并准备从文件读取
}                            // 更多字节, 将它们存入缓冲区
in.close();                 // 当使用完时, 一定要关闭信道和流
out.close();
fin.close();                // 请注意, 关闭FileChannel 并不会
fout.close();               // 自动地关闭底层的流
```

FileChannel 具有特殊的 `transferTo()` 和 `transferFrom()` method, 让从 FileChannel 传送指定数量的字节到其他指定的信道或从其他的信道到 FileChannel 变得特别容易 (而且在许多操作系统上特别有效率)。这些 method 能让我们简化前面的文件复制程序代码如下:

```
FileChannel in, out;           // 假设这些就像先前的范例中一样
                                // 被初始化
long numbytes = in.size();     // 源文件中的字节数量
in.transferTo(0, numbytes, out); // 传送那个数量到输出信道
```

可以用 `transferFrom()` 代替 `transferTo()` 写出有相同功能的程序代码 (请注意这两个 method, 可知的是它们的自变量会有不同的顺序):

```
long numbytes = in.size();
out.transferFrom(in, 0, numbytes);
```

FileChannel 具有其他信道类所没有的功能, 其中最重要的一个, 就是以“内存映射” (memory map) 文件或一部分文件的功能, 亦即取得代表文件内容的 `MappedByteBuffer` (`ByteBuffer` 的子类), 并且允许你只要通过对缓冲区做读出和写入, 就能读取及选择性地写入文件内容。以内存映射文件是消耗甚大的操作, 所以这个技巧通常只在你需要重复访问大型文件时才会有效率。内存映射还提供了另一个方法来执行先前所展示的同一文件复制操作:

```
long filesize = in.size();
ByteBuffer bb = in.map(FileChannel.MapMode.READ_ONLY, 0, filesize);
while(bb.hasRemaining()) out.write(bb);
```

由 `java.nio.channels` 所定义的信道 interface 包含了 `ByteChannel` 而没有 `CharChannel`。信道 API 是低层次的而且只提供了读取字节的 method。先前的范例都把文件当作二进制文件。使用前面介绍过的 `CharsetEncoder` 和 `CharsetDecoder` 类来在字节和字符之间转换是有可能的, 但当你想处理文本文件时, 使用 `java.io` 包的 `Reader` 和 `Writer` 类通常会比 `CharBuffer` 更容易。幸好 Channels 类定义了简单的 method, 在原有的和新的 API 之间搭起了桥梁。以下程序代码把 `Reader` 和 `Writer` 对

象封入输入和输出信道内，从输入信道读取成行的 Latin-1 文本，并通过将编码改为 UTF-8 来把它们写至输出信道。

```
ReadableByteChannel in;          // 假设这些已在其他地方初始化
WritableByteChannel out;
// 从 FileChannel 和字符集名称创建 Reader 和 Writer
BufferedReader reader=new BufferedReader(Channels.newReader(in, "ISO-8859-1"));
PrintWriter writer = new PrintWriter(Channels.newWriter(out, "UTF-8"));
String line;
while((line = reader.readLine()) != null) writer.println(line);
reader.close();
writer.close();
```

与 `FileInputStream` 及 `FileOutputStream` 类不同的是，`FileChannel` 类允许对文件内容进行随机访问。零自变量的 `position()` method 会返回文件指针（下一个要被读取的字节的位置），而一个自变量的 `position()` method 能让你把这个指针设定为你所想要的任何值，这让你能以 `java.io.RandomAccessFile` 所用的方法在文件中随机访问。这里有个范例：

```
// 假设你有个分布了数据记录的文件，而文件最后的 1024 个
// 字节是提供那些记录的位置的索引。这里的程序代码会
// 读取文件的索引，查找文件中第一次记录的位置，然后读
// 取记录

FileChannel in = new FileInputStream("test.data").getChannel(); // 信道
ByteBuffer index = ByteBuffer.allocate(1024); // 用来保存索引的缓冲区
long size = in.size(); // 文件的大小
in.position(size - 1024); // 索引的开始位置
in.read(index); // 读取索引
int record0Position = index.getInt(0); // 取得第一个索引入口
in.position(record0Position); // 移到文件中的那个位置
ByteBuffer record0 = ByteBuffer.allocate(128); // 取得缓冲区以保存数据
in.read(record0); // 最后，读取记录
```

我们在这里所考虑到的 `FileChannel` 决定性特性，就是它对所有并行访问（互斥锁）或并行写入（共享锁）锁定整个文件或一部分文件的功能（请注意，有些操作系统严格地强制执行所有的锁，而另一些则只提供了咨询性的锁定能力，它会要求程序互相协调并尝试在读取与写入共享文件的一部分之前先取得锁）。在前面的随机访问范例中，假设我们想确定在我们读取记录数据时没有其他程序正在修改它。我们可以用以下程序代码取得在那部分文件上的共享锁：

```
FileLock lock = in.lock(record0Position, // 锁定区域的开始处
                        128, // 锁定区域的长度
                        true); // 共享锁：预防并行更新
// 但允许并行读取
in.position(record0Position); // 移至索引开始处
in.read(record0); // 读取索引数据
lock.release(); // 已使用完锁，所以将它释放
```



## 客户端网络连接

New I/O API 包含了网络连接功能和文件访问功能。如果要通过网络来通信, 可以使用 `SocketChannel` 类。使用静态的 `open()` method 创建 `SocketChannel`, 然后与其他的信道对象之间进行字节的读取与写入。以下程序代码使用 `SocketChannel` 来传送 HTTP 请求到 web 服务器, 并将服务器的响应 (包括所有的 HTTP 报头) 存储在文件中。请注意 `java.net.InetSocketAddress` 的用法, 它是 `java.net.SocketAddress` 的子类, 用来告诉 `SocketChannel` 联机的对象。这些类也被引入作为 New I/O API 的一部分。

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;

// 创建一个 SocketChannel, 联机至 www.oreilly.com 的 web 服务器
SocketChannel socket =
    SocketChannel.open(new InetSocketAddress("www.oreilly.com", 80));
// 用于编码 HTTP 请求的字符集
Charset charset = Charset.forName("ISO-8859-1");
// 发送 HTTP 请求至服务器。以字符串开始, 将它封装为 CharBuffer, 编码
// 为 ByteBuffer, 然后写入 socket
socket.write(charset.encode(CharBuffer.wrap("GET / HTTP/1.0\r\n\r\n")));
// 创建 FileChannel 来存储服务器的响应
FileOutputStream out = new FileOutputStream("oreilly.html");
FileChannel file = out.getChannel();
// 取得缓冲区以在从 socket 转移到文件时保存字节
ByteBuffer buffer = ByteBuffer.allocateDirect(8192);
// 现在循环处理, 直到所有字节都已从 socket 读取并写入文件
while(socket.read(buffer) != -1 || buffer.position() > 0) { // 做完了吗?
    buffer.flip(); // 准备从缓冲区读取字节并写入文件
    file.write(buffer); // 写入一些或全部字节到文件
    buffer.compact(); // 丢弃已被写入的字节
}
socket.close(); // 关闭 socket 信道
file.close(); // 关闭文件信道
out.close(); // 关闭底层文件
```

另一个创建 `SocketChannel` 的方法, 就是使用无自变量版本的 `open()`, 它会创建未联机的信道。这让你能调用 `socket()` method 来取得底层的 socket, 依所需配置 socket 并以 `connect` method 联机至所要的主机。例如:

```
SocketChannel sc = SocketChannel.open(); // 开启一个未联机的 socket 信道
Socket s = sc.socket(); // 取得底层的 java.net.Socket
s.setSoTimeout(3000); // 在 3 秒之后超时
// 现在将 socket 信道联机至所要的主机和端口
sc.connect(new InetSocketAddress("www.davidflanagan.com", 80));

ByteBuffer buffer = ByteBuffer.allocate(8192); // 创建缓冲区
try { sc.read(buffer); } // 试着从 socket 读取
```

```

catch(SocketTimeoutException e) { // 在这里捕捉超时
    System.out.println("The remote computer is not responding.");
    sc.close();
}

```

除了SocketChannel类之外，java.nio.channels包还定义了用datagram代替socket来进行网络连接的DatagramChannel。

New I/O API的最强大特性之一，就是信道（例如SocketChannel和DatagramChannel）可以被用在非阻塞模式中。我们会在稍后的章节中看到范例。

## 服务器端网络连接

java.net包针对客户端和服务端之间的通信定义了Socket，并定义了由服务器使用的ServerSocket来监听并接受来自客户端的连接。java.nio.channels包也类似于此：它对数据转移定义了SocketChannel类，对接受联机定义了ServerSocketChannel类。ServerSocketChannel是个特殊的信道，因为它没有实现ReadableByteChannel或WritableByteChannel。它没有read()和write() method，但具有用于接受客户端连接和取得SocketChannel的accept() method，并通过SocketChannel与客户端通信。这有段程序代码，是针对简单、单一线程的服务器，它会监听8000端口的联机并回复当前时间给所有联机的客户端：

```

import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;

public class DateServer {
    public static void main(String[] args) throws java.io.IOException {
        // 取得 CharsetEncoder 以编码要送至客户端的文字
        CharsetEncoder encoder = Charset.forName("US-ASCII").newEncoder();

        // 创建一个新的 ServerSocketChannel，并将它绑定至 8000 端口
        // 请注意，这必须使用底层的 ServerSocket 来完成
        ServerSocketChannel server = ServerSocketChannel.open();
        server.socket().bind(new java.net.InetSocketAddress(8000));

        for(;;) { // 此服务器会永远运行
            // 等待客户端连接
            SocketChannel client = server.accept();
            // 以字符串形式取得当前日期与时间
            String response = new java.util.Date().toString() + "\r\n";
            // 封装、编码并传送字符串至客户端
            client.write(encoder.encode(CharBuffer.wrap(response)));
            // 与客户端中断连接
            client.close();
        }
    }
}

```

## 非阻塞式 I/O

前面的 `DateServer` 类是个简单的网络服务器。因为它没有维护与任何客户端的连接，所以绝对不会需要同时与各个客户端通信，而且使用中的 `SocketChannel` 绝不会超过一个。然而，实际上服务器必须要能同时与多个客户端通信。`java.io`和`java.net` API 只允许阻塞式 I/O，所以用这些 API 编写的服务器都必须对各个客户端使用独立的线程。对于具有许多客户端的大型服务器来说，这个方法并不恰当。为了解决这个问题，`New I/O` API 允许大部分的信道（但不包含 `FileChannel`）使用非阻塞式模式，并且允许单一线程管理所有悬置的连接。这是用 `Selector` 对象完成的，它会记录一组已注册的信道，并且可以将其冻结直到一个或多个信道已准备好做 I/O，如以下程序代码所示。这个范例比本章大部分的范例都长，但它是完整的运作中的服务器类，它管理了一个 `ServerSocketChannel` 以及任意数量通过单一 `Selector` 对象与客户端的 `SocketChannel` 连接。

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.*;           // 针对 Set 和 Iterator

public class NonBlockingServer {
    public static void main(String[] args) throws IOException {

        // 取得你所需的字符编码器和译码器
        Charset charset = Charset.forName("ISO-8859-1");
        CharsetEncoder encoder = charset.newEncoder();
        CharsetDecoder decoder = charset.newDecoder();

        // 配置缓冲区以供与客户端通信
        ByteBuffer buffer = ByteBuffer.allocate(512);

        // 此程序代码中的所有信道都会是使用非阻塞模式
        // 因此创建一个 Selector 对象，它会在监控所有的
        // 信道时冻结，并且只有在一个或多个信道准备好做
        // I/O 之类的事情时停止冻结
        Selector selector = Selector.open();

        // 创建一个新的 ServerSocketChannel，并将它连接至 8000 端口
        // 请注意，这必须使用底层 ServerSocket 来完成
        ServerSocketChannel server = ServerSocketChannel.open();
        server.socket().bind(new java.net.InetSocketAddress(8000));
        // 让 ServerSocketChannel 成为非阻塞模式
        server.configureBlocking(false);
        // 现在使用 Selector 来注册（但请注意，register() 会在信道上被
        // 调用，而不是在 selector 对象上）
        // SelectionKey 代表了此信道与此 Selector 的注册
        SelectionKey serverkey = server.register(selector,
                                                    SelectionKey.OP_ACCEPT);
```



```
for(;;) { // 主要的服务器循环。服务器会永远运行
    // 此调用会冻结，直到在其中一个已注册的信道
    // 上有活动。这是非阻塞式 I/O 中的关键 method
    selector.select();

    // 取得一个 java.util.Set，它包含了所有已准备好做 I/O 的
    // 信道的 SelectionKey 对象
    Set keys = selector.selectedKeys();

    // 使用 java.util.Iterator 来逐一处理被选定的键
    for(Iterator i = keys.iterator(); i.hasNext(); ) {
        // 取得 set 中的下一个 SelectionKey，并将它从 set 中
        // 移除。它必须被明确地移除，否则就会被下一个对
        // select() 的调用返回
        SelectionKey key = (SelectionKey) i.next();
        i.remove();

        // 检查这个键是否是在你注册 ServerSocketChannel 时所
        // 获得的 SelectionKey
        if (key == serverkey) {
            // ServerSocketChannel 上的活动代表有客户端正
            // 尝试要连接至服务器
            if (key.isAcceptable()) {
                // 接受客户端连接并取得一个 SocketChannel 来
                // 与客户端通信
                SocketChannel client = server.accept();
                // 让客户端信道成为非阻塞模式
                client.configureBlocking(false);
                // 现在用 Selector 对象来注册，告诉
                // 它你想知道什么时候会有数据从这个通
                // 道被读出
                SelectionKey clientkey =
                    client.register(selector, SelectionKey.OP_READ);
                // 添加一些客户端状态到键上。当你与客户端交谈时
                // 就会使用到这个状态
                clientkey.attach(new Integer(0));
            }
        }
        else {
            // 如果从键所组成的 Set 中取得的键不是
            // ServerSocketChannel 键，那么它就一定是代
            // 表其中一个客户端连接的键
            // 从键取得信道
            SocketChannel client = (SocketChannel) key.channel();

            // 如果已执行到这，信道中就应该会有数据可读
            // 取，但要再做检查
            if (!key.isReadable()) continue;

            // 现在从客户端读取字节。假设客户端的所有字节
            // 都在一个读取操作中
            int bytesread = client.read(buffer);

            // 如果 read() 返回 -1，就是指示流的结尾，这
            // 代表客户端已中断连接，所以要解除已选取键的
```

```
// 注册并关闭信道
if (bytesread == -1) {
    key.cancel();
    client.close();
    continue;
}
// 否则，将字节译码为请求字符串
buffer.flip();
String request = decoder.decode(buffer).toString();
buffer.clear();
// 现在以请求字符串为基础来回客户端
if (request.trim().equals("quit")) {
    // 如果请求是“quit”，就送出最后的信息
    // 关闭信道并解除对 SelectionKey 的注册
    client.write(encoder.encode(CharBuffer.wrap("Bye.")));
    key.cancel();
    client.close();
}
else {
    // 否则，发送由此请求的序号加上大写版
    // 本的请求字符串所组成的响应字符串。请注
    // 意，记录序号的方法是通过“添加”
    // 对象到 SelectionKey 并每次递增

    // 从 SelectionKey 取得序号
    int num = ((Integer)key.attachment()).intValue();
    // 针对响应字符串
    String response = num + ": " +
        request.toUpperCase();
    // 封装、编码并写入响应字符串
    client.write(encoder.encode(CharBuffer.wrap(response)));
    // 添加递增的序号至键
    key.attach(new Integer(num+1));
}
}
}
}
```

非阻塞式 I/O 对于网络服务器写入最有用，它对于同时有多个网络连接悬置的客户端也很有用。例如，考虑 web 浏览器下载网页以及那个页面同时引用到的图像。非阻塞式 I/O 的另一个有趣用法就是执行非阻塞式 socket 连接操作。这个概念就是你可以要求 `SocketChannel` 建立连接至远程主机，然后在底层 OS 设定跨网络的连接时做其他事（例如建立 GUI）。之后，如果连接还没建立好，你就进行 `select()` 调用以冻结，直到连接已被建立。针对非阻塞式连接的程序代码看起来就像这样：

```
// 创建新的、未联机的 SocketChannel。把它设定为非阻塞模式，使用新的 Selector 加以注册，然后告诉它进行连接。connect 调用会返回，而不是等待网络连接被完全建立
```

```
Selector selector = Selector.open();
SocketChannel channel = SocketChannel.open();
channel.configureBlocking(false);
channel.register(selector, SelectionKey.OP_CONNECT);
channel.connect(new InetSocketAddress(hostname, port));

// 现在在连接建立期间做其他的事
// 例如, 在这建立 GUI

// 如果现在有需要就冻结, 直到 SocketChannel 已准备好联机
// 由于你已用这个 selector 注册了唯一的信道, 所以不必查看键 set,
// 你知道哪个信道已准备好
while(selector.select() == 0) /* 空循环 */;

// 此调用对于完成非阻塞式连接是必需的
channel.finishConnect();

// 最后, 关闭 selector, 这会解除信道的注册
selector.close();
```

## XML

Java 1.4 和 Java 5.0 已为 Java 平台增加了强大的 XML 处理特性:

### org.xml.sax

此包以及它的两个子包定义了实际标准的 SAX API (SAX 代表 Simple API for XML)。SAX 是个事件驱动、XML 解析的 API: 在解析 XML 文件时, SAX 解析器会调用指定 ContentHandler 对象 (以及其他相关的处理程序对象) 的 method。文件的结构和内容会由 method 调用来完整描述。要存储任何的状态或执行任何适当的动作是取决于 ContentHandler 实现。此包包括了针对 SAX 2 API 的类以及针对 SAX 1 的过时类。

### org.w3c.dom

此包定义了一些 interface, 以树形结构来表现 XML 文件。Document Object Model (DOM) 是 World Wide Web Consortium (W3C) 的建议 (实质上是标准)。DOM 解析器会读取 XML 文件并将它转换成由节点组成的树, 那代表了文件的完整内容。一旦文件的树形结构被建立, 程序就可以随意查看并操作它。Java 1.4 包含了 Level 2 DOM 的核心模块, 而 Java 5.0 包含了 Level 3 DOM 的核心、事件以及加载/存储模块。

### javax.xml.parsers

此包提供了用于实例化 SAX 和 DOM 解析器以解析 XML 文件的高层次 interface。



javax.xml.transform

此包以及它的子包定义了使用 XSLT 标准转换 XML 文件内容和表达方法的 Java API。

javax.xml.validation

这个 Java 5.0 包对验证模式的 XML 文件提供了支持。实现必须支持 W3C XML Schema 标准，而且也要支持其他的模式类型。

javax.xml.xpath

此包也是 Java 5.0 中新出现的，支持 XPath 的判定以选择 XML 文件中的节点。

使用这些包的范例会出现在下面的章节中。

## 使用 SAX 解析 XML

使用 SAX 解析 XML 文件的第一步就是取得 SAX 解析器。如果你有自己的 SAX 解析器实现，就只要实例化适当的解析器类即可。但是，通常较简单的方式是使用 javax.xml.parsers 包来实例化由 Java 实现所提供的 SAX 解析器。程序代码看起来就像这样：

```
import javax.xml.parsers.*;

// 取得用来创建 SAX 解析器的 factory 对象
SAXParserFactory parserFactory = SAXParserFactory.newInstance();

// 设定 factory 对象来指定它所创建的解析器的属性
parserFactory.setValidating(true);
parserFactory.setNamespaceAware(true);

// 现在创建一个 SAXParser 对象
SAXParser parser = parserFactory.newSAXParser(); // 可能会抛出异常
```

SAXParser 是个围绕于 org.xml.sax.XMLReader 类的简单封装程序。一旦你取得它，就如前面的程序代码，只要调用各种 parse() method 就可以解析文件了。其中有一些 method 使用了过时的 SAX 1 HandlerBase 类，而其他的使用了当前的 SAX 2 org.xml.sax.helpers.DefaultHandler 类。DefaultHandler 类提供了所有 ContentHandler、ErrorHandler、DTDHandler 以及 EntityResolver interface 的空实现，这些都是 SAX 解析器在解析 XML 文件时可以调用的 method。通过使用 DefaultHandler 的子类并定义你自己关心的 method，就可以执行任何必需的动作来响应由解析器产生的 method 调用。以下的程序代码显示了使用 SAX 来解析 XML 文件的 method，并判定出现在文件中的 XML 元素数量以及出现在那些元素中的一般文字的字符数量（也许排除了“可忽略的空白”）：

```

import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class SAXCount {
    public static void main(String[] args)
        throws SAXException, IOException, ParserConfigurationException
    {
        // 创建一个解析器 factory 并用它来产生解析器
        SAXParserFactory parserFactory = SAXParserFactory.newInstance();
        SAXParser parser = parserFactory.newSAXParser();
        // 这是你所要解析的文件的名称
        String filename = args[0];
        // 实例化 DefaultHandler 子类来进行计算
        CountHandler handler = new CountHandler();
        // 启动解析器。它会读取文件并调用处理程序的 method
        parser.parse(new File(filename), handler);
        // 当完成时，报告由处理程序对象所存储的结果
        System.out.println(filename + " contains " + handler.numElements +
            " elements and " + handler.numChars +
            " other characters ");
    }

    // 这个内部类扩展了 DefaultHandler 来计算 XML 文件
    // 中的元素和文字数量，并把结果存储在公共字段中。
    // 你还可以覆盖很多其他的 DefaultHandler method，但你
    // 只需要这些
    public static class CountHandler extends DefaultHandler {
        public int numElements = 0, numChars = 0; // 在这储存计数
        // 此 method 会在解析器遇到任何 XML 元素的开始标记时
        // 被调用。忽略这些自变量但计算元素
        public void startElement(String uri, String localname, String qname,
            Attributes attributes) {
            numElements++;
        }

        // 此 method 会在遇到元素中的任意一个一般文字时被调用
        // 仅只计算那个文字中的字符数量
        public void characters(char[] text, int start, int length) {
            numChars += length;
        }
    }
}

```

## 使用 DOM 解析 XML

DOM API 与 SAX API 差异极大。虽然 SAX 在扫描 XML 文件时是很有效率的方式，但它不太适合想要修改文件的程序。DOM 解析器不是把 XML 文件转换为一连串的 method 调用，而是把文件转换为 `org.w3c.dom.Document` 对象，那是由 `org.w3c.dom.Node`

对象组成的树。将全部XML文件转换为树形结构后能提供对整个文件的随机访问,但会消耗大量的内存。

在DOM API中,文件树中的每个节点实现了Node interface和类型特有的subinterface(DOM文件中最常见的节点类型是Element和Text节点)。当解析器对文件的解析完成时,你的程序就可以使用Node及其subinterface的各种method来查看及操作那个树。以下程序代码使用JAXP来取得DOM解析器(在JAXP的定义中,它被称为DocumentBuilder)。它接下来会解析XML文件并从中建立一个文件树。然后,它查看文件树来搜索<sect1>元素并输出每个<title>的内容。

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class GetSectionTitles {
    public static void main(String[] args)
        throws IOException, ParserConfigurationException,
            org.xml.sax.SAXException
    {
        // 创建一个 factory 对象以供创建 DOM 解析器并加以设定
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setIgnoringComments(true); // 我们想忽略注释
        factory.setCoalescing(true); // 将CDATA转换为Text节点
        factory.setNamespaceAware(false); // 无命名空间:这是默认的
        factory.setValidating(false); // 不验证DTD:这也是默认的

        // 现在使用 factory 来创建 DOM 解析器,也就是 DocumentBuilder
        DocumentBuilder parser = factory.newDocumentBuilder();

        // 解析文件并建立文件树来代表其内容
        Document document = parser.parse(new File(args[0]));

        // 查询文件所包含的所有 <sect1> 元素的列表
        NodeList sections = document.getElementsByTagName("sect1");
        // 逐一处理那些 <sect1> 元素,一次处理一个
        int numSections = sections.getLength();
        for(int i = 0; i < numSections; i++) {
            Element section = (Element)sections.item(i); // 一个 <sect1>
            // 每个 <sect1> 的第一个子元素应该是 <title> 元
            // 素,但可能会先有一些空白 Text 节点,所
            // 以逐一处理直到找到第一个子元素
            Node title = section.getFirstChild();
            while(title != null && title.getNodeType() != Node.ELEMENT_NODE)
                title = title.getNextSibling();
            // 列出此元素的 Text 节点所包含的文字
            if (title != null)
                System.out.println(title.getFirstChild().getNodeValue());
        }
    }
}
```



## 转换 XML 文件

javax.xml.transform包定义了用来创建Transformer对象的TransformerFactory类。Transformer可以将文件从Source表达方式转换为Result表达方式，并在处理过程中选择性地对文件内容应用XSLT转换。有三个子包定义了Source和Result interface的具体实现，让文件能在这三种表达方式之间转换：

javax.xml.transform.stream

将文件当成由 XML 文本所组成的流来表达。

javax.xml.transform.dom

将文件当成 DOM Document 树来表达。

javax.xml.transform.sax

将文件当作一连串的 SAX method 调用来表达。

以下程序代码说明这些包将文件的表达方式从DOM Document树转换为XML文本所组成的流的方法。这段程序代码有个有趣的特性就是，它没有通过解析文件来建立Document树，而是从头开始建立。

```
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class DOMToStream {
    public static void main(String[] args)
        throws ParserConfigurationException,
               TransformerConfigurationException,
               TransformerException
    {
        // 创建 DocumentBuilderFactory 和 DocumentBuilder
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        // 然而不解析 XML 文件，只创建可以由你自己增加内容的空文件
        Document document = db.newDocument();

        // 现在使用 DOM method 来建立文件树
        Element book = document.createElement("book"); // 创建新元素
        book.setAttribute("id", "javanut4");           // 给它一个属性
        document.appendChild(book);                     // 放入文件
        for(int i = 1; i <= 3; i++) {                   // 增加更多元素
            Element chapter = document.createElement("chapter");
            Element title = document.createElement("title");
            title.appendChild(document.createTextNode("Chapter " + i));
            chapter.appendChild(title);
        }
    }
}
```

```

        chapter.appendChild(document.createElement("para"));
        book.appendChild(chapter);
    }

    // 现在创建一个 TransformerFactory 并用它来创建 Transformer 对象, 把
    // 我们的 DOM 文件转换为由 XML 文本组成的流
    // newTransformer() 没有自变量意味着没有 XSLT 样式表
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer transformer = tf.newTransformer();

    // 创建用于转换的 Source 和 Result 对象
    DOMSource source = new DOMSource(document);           // DOM 文件
    StreamResult result = new StreamResult(System.out);    // 成为 XML 文本

    // 最后, 进行转换
    transformer.transform(source, result);
}
}

```

javax.xml.transform 最有趣的用法牵涉到 XSLT 样式表。XSLT 是个复杂但强大的 XML 文法, 它描述了 XML 文件内容被转换为另一种形式 (例如, XML、HTML 或一般文字) 的方法。XSLT 样式表的教学超出了本书的范围, 但以下程序代码 (只包含 6 个关键行) 说明了将这样的样式表 (它本身是个 XML 文件) 应用至另一个 XML 文件, 并将最后产生的文件写入流的方法:

```

import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class Transform {
    public static void main(String[] args)
        throws TransformerConfigurationException,
            TransformerException
    {
        // 取得针对输入、样式表以及输出的 Source 和 Result 对象
        StreamSource input = new StreamSource(new File(args[0]));
        StreamSource stylesheet = new StreamSource(new File(args[1]));
        StreamResult output = new StreamResult(new File(args[2]));

        // 创建一个 transformer 并执行转换
        TransformerFactory tf = TransformerFactory.newInstance();
        Transformer transformer = tf.newTransformer(stylesheet);
        transformer.transform(input, output);
    }
}

```

## 验证 XML 文件

javax.xml.validation 包能让你针对模式验证 XML 文件。从 javax.xml.parsers 包取得的 SAX 和 DOM 解析器可以在解析过程中对 DTD 执行验证，但此包将验证与解析分开，并且也对任意的模式类型提供了支持。所有的实现都必须支持 W3C XML Schema 并且允许支持其他的模式类型，例如 RELAX NG。

如果使用包，则要以 SchemaFactory instance 开始——那是针对指定模式类型的解析器，使用此解析器将模式文件解析为 Schema 对象。从 Schema 取得 Validator，然后使用 Validator 来验证你的 XML 文件。文件被指定为 SAXSource 或 DOMSource 对象。你可以从 javax.xml.transform 取回这些类。

如果文件是有效的，Validator 对象的 validate() method 就会正常返回；如果文件不是有效的，validate() 就会抛出 SAXException。你可以安装针对 Validator 的 org.xml.sax.ErrorHandler 对象来对会造成异常的各种验证错误提供一些控制。

```
import javax.xml.XMLConstants;
import javax.xml.validation.*;
import javax.xml.transform.sax.SAXSource;
import org.xml.sax.*;
import java.io.*;

public class Validate {
    public static void main(String[] args) throws IOException {
        File documentFile = new File(args[0]); // 第一个自变量是文件
        File schemaFile = new File(args[1]);   // 第二个自变量是模式

        // 取得解析器来解析 W3C 模式。请注意 javax.xml 包的用法
        // 此包只包含了一个由常量组成的类
        SchemaFactory factory =
            SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);

        // 现在解析模式文件来创建 Schema 对象
        Schema schema = null;
        try { schema = factory.newSchema(schemaFile); }
        catch(SAXException e) { fail(e); }

        // 从 Schema 取得 Validator 对象
        Validator validator = schema.newValidator();

        // 取得针对文件的 SAXSource 对象
        // 我们也可以在这儿使用 DOMSource
        SAXSource source =
            new SAXSource(new InputSource(new FileReader(documentFile)));

        // 现在验证文件
        try { validator.validate(source); }
        catch(SAXException e) { fail(e); }
```



```

        System.err.println("Document is valid");
    }

    static void fail(SAXException e) {
        if (e instanceof SAXParseException) {
            SAXParseException spe = (SAXParseException) e;
            System.err.printf("%s:%d:%d: %s%n",
                              spe.getSystemId(), spe.getLineNumber(),
                              spe.getColumnNumber(), spe.getMessage());
        }
        else {
            System.err.println(e.getMessage());
        }
        System.exit(1);
    }
}

```

## 求 XPath 表达式的值

XPath 是用来引用 XML 文件中的特定节点的语言。例如，XPath 表达式 “//section/title/text()” 会引用文件中的任意深度的 <section> 元素中的 <title> 元素内的文字。关于 XPath 语言的完整说明超出了本书的范围。Java 5.0 中新出现的 javax.xml.xpath 包提供了一个方法来找出文件中与 XPath 表达式相匹配的所有节点。

```

import javax.xml.xpath.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class XPathEvaluator {
    public static void main(String[] args)
        throws ParserConfigurationException, XPathExpressionException,
               org.xml.sax.SAXException, java.io.IOException
    {
        String documentName = args[0];
        String expression = args[1];

        // 将文件解析为 DOM 树
        // XPATH 也可以与 SAXInputSource 一起使用
        DocumentBuilder parser =
            DocumentBuilderFactory.newInstance().newDocumentBuilder();
        Document doc = parser.parse(new java.io.File(documentName));

        // 取得 XPath 对象来求表达式的值
        XPath xpath = XPathFactory.newInstance().newXPath();

        System.out.println(xpath.evaluate(expression, doc));

        // 或求表达式的值以取得所有和节点相匹配的 DOM NodeList, 然后
        // 逐一处理各个节点直至最后
        NodeList nodes = (NodeList)xpath.evaluate(expression, doc,
                                                    XPathConstants.NODESET);
    }
}

```

```
        for(int i = 0, n = nodes.getLength(); i < n; i++) {
            Node node = nodes.item(i);
            System.out.println(node);
        }
    }
}
```

## 类型、反射与动态加载

java.lang.Class类表达了Java中的数据类型还有java.lang.reflect包中的类,它给Java程序提供了自我评测(或自反射)的能力。Java类可以考虑它自己或任何其他类,并判定它的超类及它定义了哪些method等。

### Class 对象

有好几种方法可以在Java中取得Class对象:

```
// 取得任意对象o的Class
Class c = o.getClass();

// 以各种预先定义的常量取得基本类型的Class对象
c = Void.TYPE;           // 特殊的“无返回值”类型
c = Byte.TYPE;           // 代表字节的Class对象
c = Integer.TYPE;        // 代表整数的Class对象
c = Double.TYPE;         // 诸如此类。类似的还有 Short、Character、Long、Float

// 将类直接量表达为类型名称后面接着“.class”
c = int.class;           // 与 Integer.TYPE 相同
c = String.class;        // 与 "dummystring".getClass() 相同
c = byte[].class;        // 字节数组的类型
c = Class[][].class;     // 由 Class 对象组成的数组所组成的数组类型
```

### Class 上的反射 (Reflection)

一旦有了Class对象,你就可以用它执行一些有趣的反射操作:

```
import java.lang.reflect.*;

Object o;                // 某个要研究的未知对象
Class c = o.getClass();   // 取得其类型

// 如果它是数组,就找出它的基类型
while (c.isArray()) c = c.getComponentType();

// 如果c不是基本类型,就输出它的类层次
if (!c.isPrimitive()) {
    for(Class s = c; s != null; s = s.getSuperclass())
        System.out.println(s.getName() + " extends");
}
```

```
// 尝试创建属于 c 的新实例，这会需要无自变量的构造函数
Object newobj = null;
try { newobj = c.newInstance(); }
catch (Exception e) {
    // 处理 InstantiationException、IllegalAccessException
}

// 查看类是否有个名称为 setText 的 method，它会接收一个 String
// 如果有，就用字符串自变量来调用它
try {
    Method m = c.getMethod("setText", new Class[] { String.class });
    m.invoke(newobj, new Object[] { "My Label" });
} catch (Exception e) { /* 在这处理异常 */ }

// 这些是 Java 5.0 中的 varargs method，所以语法会更为清楚
// 寻找并调用名称为 "put" 的 method，它会接收两个 Object 自变量
try {
    Method m = c.getMethod("add", Object.class, Object.class);
    m.invoke(newobj, "key", "value");
} catch (Exception e) { System.out.println(e); }

// 在 Java 5.0 中，我们可以在枚举类型和常量上使用反射 (reflection)
Class<Thread.State> ts = Thread.State.class; // Thread.State 类型
if (ts.isEnum()) { // 如果是枚举类型
    Thread.State[] constants = ts.getEnumConstants(); // 取得它的常量
}
try {
    Field f = ts.getField("RUNNABLE"); // 取得名称为 "RUNNABLE" 的字段
    System.out.println(f.isEnumConstant()); // 它是个列举常量吗?
}
catch (Exception e) { System.out.println(e); }

// JVM 会在运行时丢弃泛型信息，但它会由于编译器而被存储在类文件
// 中，并且可以通过反射来访问
try {
    Class map = Class.forName("java.util.Map");

    TypeVariable<?>[] typevars = map.getTypeParameters();
    for (TypeVariable<?> typevar : typevars) {
        System.out.print(typevar.getName());
        Type[] bounds = typevar.getBounds();
        if (bounds.length > 0) System.out.print(" extends ");
        for (int i = 0; i < bounds.length; i++) {
            if (i > 0) System.out.print(" & ");
            System.out.print(bounds[i]);
        }
        System.out.println();
    }
}
catch (Exception e) { System.out.println(e); }

// 在 Java 5.0 中，反射也可被用在 annotation 类型上，并决定
// 运行时可见注释的值
```



```

Class<?> a = Override.class; // annotation 类
if (a.isAnnotation()) {      // 这是个 annotation 类型吗?
    // 寻找一些 meta-annotation
    java.lang.annotation.Retention retention =
        a.getAnnotation(java.lang.annotation.Retention.class);
    if (retention != null)
        System.out.printf("Retention: %s\n", retention.value());
}

```

## 动态类加载

Java 中的 Class 也为动态类加载提供了简单的机制。但是，对于在动态类加载上的更加完全的控制，你应该使用 `java.lang.ClassLoader` 对象，通常是 `java.net.URLClassLoader`。这个技术很有用，例如，当你想加载的类是被命名在配置文件中而不是被编写在程序中的时候。

```

// 动态地加载名称指定于配置文件中的类
String classname =                // 查询类的名称
    config.getProperty("filterclass", // 属性名称
        "com.davidflanagan.filters.Default"); // 默认值

try {
    Class c = Class.forName(classname); // 动态地加载类
    Object o = c.newInstance();         // 动态地将它实例化
} catch (Exception e) { /* 处理异常 */ }

```

前面的程序代码只有在被加载的类在类路径中才行得通。如果情况并非如此，你可以创建自定义的 `ClassLoader` 对象从你自己指定的路径（或 URL）加载类：

```

import java.net.*;
String classdir = config.getProperty("filterDirectory"); // 查询类路径
try {
    ClassLoader loader = new URLClassLoader(new URL[] { new URL(classdir) });
    Class c = loader.loadClass(classname);
}
catch (Exception e) { /* 处理异常 */ }

```

## 动态代理服务器

`java.lang.reflect` 包的 `Proxy` 类和 `InvocationHandler` interface 是在 Java 1.3 中增加的。`Proxy` 是个强大但不常使用的类，它能让你动态地创建实现一个特定 interface 或一组 interface 的新类或实例。它也分派了对 `InvocationHandler` 对象的 interface method 调用。

## 对象持久性保存

Java 平台为对象持久保存 (persistence) 提供了两个机制：存储对象状态的能力，以便对象可在稍后被重建。这两个机制都牵涉到序列化，其中第二个是特别针对 JavaBean 的。

### 序列化

java.io 包的最重要特性之一就是对象序列化：把对象转换成字节流，以在稍后反序列化为原始对象的副本。以下程序代码显示如何使用序列化把对象存储至文件并在稍后将它读回：

```
Object o; // 我们要序列化的对象，它必须实现 Serializable
File f;   // 用来存储数据的文件

try {
    // 将对象序列化
    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(f));
    oos.writeObject(o);
    oos.close();

    // 将对象读回
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream(f));
    Object copy = ois.readObject();
    ois.close();
}
catch (IOException e) { /* 处理输入 / 输出异常 */ }
catch (ClassNotFoundException cnfe) { /* readObject() 可能会抛出这个 */ }
```

先前的范例序列化至文件，但请记住，你可以将已序列化对象写入任一种流类型。因此，你可以把对象写入字节数组，然后把它从字节数组读回，创建对象的副本。你可以把对象的字节写入压缩流甚至将字节跨网络连接发送至另一个程序的流！

### JavaBeans 持久保存

Java 1.4 引入了序列化机制是为了要配合 JavaBeans 组件使用。java.io 序列化是通过存储对象的内部字段的内部状态来完成。另一方面，java.beans 持久保存是通过将 bean 的状态存储为对定义于类内的公共 method 的一连串调用。由于它是以公共 API 为基础，而不是内部状态，所以 JavaBeans 持久保存机制允许相同 API 不同实现之间的互通，以更完善的方式来处理版本差异，并且适合已序列化对象的长期存储。

bean 以及任一个 descendant bean 或其他被 java.beans.XMLDecoder 序列化的对象，可以用 java.beans.XMLDecoder 来反序列化。这些类会对指定的流做写入和读取，但它们本身并不是流类。这里是可用来对 bean 编码的方式：

```
// 创建 JavaBean 并设定它的一些属性
javax.swing.JFrame bean = new javax.swing.JFrame("PersistBean");
bean.setSize(300, 300);
// 现在将它已编码的格式存储在文件 bean.xml 中
BufferedOutputStream out = // 创建输出流
    new BufferedOutputStream(new FileOutputStream("bean.xml"));
XMLEncoder encoder = new XMLEncoder(out); // 创建流的编码器
encoder.writeObject(bean); // 对 bean 编码
encoder.close(); // 关闭编码器和流
```

这里是相对应的程序代码，用来将 bean 从其已序列化的形式中解码：

```
BufferedInputStream in = // 创建输入流
    new BufferedInputStream(new FileInputStream("bean.xml"));
XMLDecoder decoder = new XMLDecoder(in); // 创建流的译码器
Object b = decoder.readObject(); // 对 bean 解码
decoder.close(); // 关闭译码器和流
bean = (javax.swing.JFrame) b; // 将 bean 转换为适当的类型
bean.setVisible(true); // 开始使用它
```

## 安全性

java.security 包定义了一些与 Java 访问控制结构有关的类，这会在第六章做更详细的讨论。这些类能让 Java 程序在受限制的环境中运行未被信任的程序代码，在那个环境中，程序代码不会造成什么损害。虽然这些是重要的类，但你很少需要使用它们。较有趣的类是用于消息摘要和数字签名的那些类，在以下章节会对它们作示范。

## 消息摘要

消息摘要 (message digest) 是个值，也就是所谓的加密校验和 (cryptographic checksum) 或安全散列 (secure hash)，那是根据一连串的字节的计算得到的。摘要的长度通常会远小于计算时所依据的数据的长度。当传送数据 (消息) 时，你可以把消息摘要一起传出去。消息的接收者接着就可以重新计算已收到数据上的消息摘要，并通过比较计算得到的摘要与收到的摘要来判定消息或摘要是否在传输过程中损毁或被篡改。我们在本章的前面讨论流时，已看到了一个计算消息摘要的方法。类似的技术也可以用来计算非流形式的二进制数据的消息摘要：

```
import java.security.*;

// 取得一个对象并使用“安全散列算法 (Secure Hash Algorithm)”来计算消息摘要
// 这个方法可能会抛出 NoSuchAlgorithmException
MessageDigest md = MessageDigest.getInstance("SHA");

byte[] data, data1, data2, secret; // 一些已在其他地方被初始化的字节数组
```



```
// 创建针对单一字节数组的摘要
byte[] digest = md.digest(data);

// 创建针对几大块数据的摘要
md.reset();           // 选择性的：会由 digest() 自动调用
md.update(data1);      // 处理第一块数据
md.update(data2);      // 处理第二块数据
digest = md.digest();  // 计算摘要

// 创建加密摘要 (keyed digest)，如果你知道加密字节 (secret byte) 就能做验证
md.update(data);       // 要与摘要一起传送的数据
digest = md.digest(secret); // 加入加密字节并计算摘要

// 用像这样的方式验证摘要
byte[] receivedData, receivedDigest; // 我们所收到的数据和摘要
byte[] verifyDigest = md.digest(receivedData); // 对已收到的数据做摘要
// 比较计算得来的摘要和收到的摘要
boolean verified = java.util.Arrays.equals(receivedDigest, verifyDigest);
```

## 数字签名

数字签名 (digital signature) 结合了消息摘要算法和公共密钥密码学 (public-key cryptography)。消息的传送者 Alice 可以计算消息的摘要，然后用她的私用密钥 (private key) 来加密摘要。接着她把消息和已加密的摘要传送给接收者 Bob。Bob 知道 Alice 的公共密钥 (毕竟那是公开的)，所以他可以用它来解密摘要并验证那个消息没有被篡改。在执行验证时，Bob 也知道摘要是用 Alice 的私用密钥加密的，因为他能成功地用 Alice 的公共密钥来对摘要解密。由于 Alice 是唯一知道自己的私用密钥的人，所以信息一定是来自 Alice。数字签名就是因此得名，就像笔和纸的签名一样，它可以验证文件或信息的来源。但是，与笔和纸的签名所不同的是，要伪造数字签名虽不是不可能，但难度非常高，这不仅是剪贴到另一份文件而已。

Java 使得创建数字签名变得很容易。但是，为了要创建数字签名，你需要有 `java.security.PrivateKey` 对象。假设你的系统中已有密钥库 (keystore) (请参阅第八章的 `keytool` 说明文件)，就可以用如下的程序代码来取得一个密钥：

```
// 这里是我们所需要的一些基本数据
File homedir = new File(System.getProperty("user.home"));
File keyfile = new File(homedir, ".keystore"); // 或从配置文件读取
String filepass = "KeyStore password"         // 用于整个文件的密码
String signer = "david";                      // 从配置文件读取
String password = "No one can guess this!";    // 最好对此做提示
PrivateKey key; // 这是我们在密钥库中寻找的密钥

try {
    // 取得 KeyStore 对象，然后将数据放入
    KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
    keystore.load(new BufferedInputStream(new FileInputStream(keyfile)),
        filepass.toCharArray());
}
```

```
// 现在得到了所需的密钥
key = (PrivateKey) keystore.getKey(signer, password.toCharArray());
}
catch (Exception e) { /* 在这处理各种异常 */ }
```

一旦有了 `PrivateKey` 对象，就可以用 `java.security.Signature` 对象来创建数字签名：

```
PrivateKey key; // 就如前面所显示的，已初始化
byte[] data; // 要被签名的数据
Signature s = // 取得对象来创建并验证签名
    Signature.getInstance("SHA1withDSA"); // 可能会抛出
// NoSuchAlgorithmException
s.initSign(key); // 将它初始化，可能会抛出 InvalidKeyException
s.update(data); // 要签名的数据，可能会抛出 SignatureException
/* s.update(data2); */ // 调用多次以声明所有的数据
byte[] signature = s.sign(); // 计算签名
```

`Signature` 对象可以验证数字签名：

```
byte[] data; // 加上签名的数据，已在其他地方初始化
byte[] signature; // 要被验证的签名，已在其他地方初始化
String signername; // 创建签名的人，已在其他地方初始化
KeyStore keystore; // 存储认证的地方，就如前面所显示的，已初始化

// 寻找此签名者的公共密钥认证
java.security.cert.Certificate cert = keystore.getCertificate(signername);
PublicKey publicKey = cert.getPublicKey(); // 从这取得公共密钥

Signature s = Signature.getInstance("SHA1withDSA"); // 或者某种其他的算法
s.initVerify(publicKey); // 用于验证的设定
s.update(data); // 说明已签名的数据
boolean verified = s.verify(signature); // 验证签名数据
```

## 已签名对象

`java.security.SignedObject` 类是个有用的公共程序，用于将数字签名与对象封装在一起。`SignedObject` 接着就可以被序列化并传送到接收者，接收者可以将它反序列化并使用 `verify()` method 来验证签名：

```
Serializable o; // 要被加上签名的对象，必须是 Serializable
PrivateKey k; // 用来签名的密钥，已在其他地方初始化
Signature s = Signature.getInstance("SHA1withDSA"); // 签名“引擎”
SignedObject so = new SignedObject(o, k, s); // 创建 SignedObject

// SignedObject 封装了对象 o。现在它可以被序列化
// 并传送到接收者

// 这里是接收者验证 SignedObject 的方法
SignedObject so; // 要被反序列化的 SignedObject
Object o; // 要从 so 中提取出来的原始对象
PublicKey pk; // 用来验证的密钥
```

```
Signature s = Signature.getInstance("SHA1withDSA"); // 验证“引擎”
if (so.verify(pk,s)) // 如果签名是有效的,
    o = so.getObject(); // 就提取被封装的对象
```

## 密码术

java.security包包括了基于密码术的类,但它不包含用于实际加密和解密的类,那是javax.crypto包的工作。此包支持对称密钥密码术,在这种密码术中,加密和解密都是使用相同的密钥,而且加密数据的传送者和接收者都必须知道这个密钥。

### 加密密钥 (secret key)

SecretKey interface 代表了加密密钥,任何加密操作的第一步都是取得适当的SecretKey。很不幸的是,JDK 所提供的keytool 程序无法产生并存储加密密钥,所以程序必须自己处理这些工作。这里有一些程序代码,显示了各种使用SecretKey对象的方法:

```
import javax.crypto.*;
import javax.crypto.spec.*;
// 使用 KeyGenerator 对象产生加密密钥

KeyGenerator desGen = KeyGenerator.getInstance("DES"); // DES 算法
SecretKey desKey = desGen.generateKey(); // 产生密钥
KeyGenerator desEdeGen = KeyGenerator.getInstance("DESEde"); // Triple DES
SecretKey desEdeKey = desEdeGen.generateKey(); // 产生密钥

// SecretKey 是不透明的密钥表示方式。使用 SecretKeyFactory 来转
// 换成具有透明性、操作性的表达方式:存储在文件中,安全地
// 传送至接收端等
SecretKeyFactory desFactory = SecretKeyFactory.getInstance("DES");
DESKeySpec desSpec = (DESKeySpec)
    desFactory.getKeySpec(desKey, javax.crypto.spec.DESKeySpec.class);
byte[] rawDesKey = desSpec.getKey();
// 对 DESEde 密钥做相同的事
SecretKeyFactory desEdeFactory = SecretKeyFactory.getInstance("DESEde");
DESEKeySpec desEdeSpec = (DESEKeySpec)
    desEdeFactory.getKeySpec(desEdeKey, javax.crypto.spec.DESEKeySpec.class);
byte[] rawDesEdeKey = desEdeSpec.getKey();

// 将密钥的原始字节转换成 SecretKey 对象
DESEKeySpec keySpec = new DESEKeySpec(rawDesEdeKey);
SecretKey k = desEdeFactory.generateSecret(keySpec);

// 对于 DES 和 DESEde 密钥,有更简单的密钥创建方法
// SecretKeySpec 实现了 SecretKey,所以用它来代表这些密钥
byte[] desKeyData = new byte[8]; // 从文件读取 8 字节的数据
byte[] tripleDesKeyData = new byte[24]; // 从文件读取 24 字节的数据
SecretKey myDesKey = new SecretKeySpec(desKeyData, "DES");
SecretKey myTripleDesKey = new SecretKeySpec(tripleDesKeyData, "DESEde");
```



## 使用密码进行加密与解密

一旦你取得适当的SecretKey对象,用于加密和解密的主要类就是Cipher。用法如下:

```
SecretKey key;          // 用先前所示的方法取得 SecretKey
byte[] plaintext;       // 要加密的数据,已在其他地方初始化

// 得到对象来执行加密或解密
Cipher cipher = Cipher.getInstance("DESede"); // Triple-DES 加密
// 初始化 cipher 对象以供加密
cipher.init(Cipher.ENCRYPT_MODE, key);
// 现在加密数据
byte[] ciphertext = cipher.doFinal(plaintext);

// 如果我们有多块数据要加密,可以这么做
cipher.update(message1);
cipher.update(message2);
byte[] ciphertext = cipher.doFinal();

// 只要倒过来解密
cipher.init(Cipher.DECRYPT_MODE, key);
byte[] decryptedMessage = cipher.doFinal(ciphertext);

// 解密多块数据
byte[] decrypted1 = cipher.update(ciphertext1);
byte[] decrypted2 = cipher.update(ciphertext2);
byte[] decrypted3 = cipher.doFinal(ciphertext3);
```

## 对流加密与解密

Cipher 类也可以配合 CipherInputStream 或 CipherOutputStream 使用,在读取与写入流数据时进行加密或解密:

```
byte[] data;                // 要加密的数据
SecretKey key;              // 以先前所示的方式初始化
Cipher c = Cipher.getInstance("DESede"); // 要执行加密的对象
c.init(Cipher.ENCRYPT_MODE, key); // 把它初始化

// 创建一个流来把字节写到文件
FileOutputStream fos = new FileOutputStream("encrypted.data");

// 创建一个流,在把字节送到那个流之前先加密
// 如果要在读取字节期间加密或解密,请参阅 CipherInputStream
CipherOutputStream cos = new CipherOutputStream(fos, c);

cos.write(data);             // 将数据加密并写入文件
cos.close();                 // 一定要记得关闭流
java.util.Arrays.fill(data, (byte)0); // 清除未被加密的数据
```

## 已加密对象

最后, `javax.crypto.SealedObject` 类提供了特别简单的方式来执行加密。此类会序列化指定的对象, 并加密最后产生的字节流。`SealedObject` 本身接着可以被序列化并传送给接收者。接收者只有在知道必要的 `SecretKey` 时才能获取原始对象:

```
Serializable o;           // 要被加密的对象, 必须是 Serializable
SecretKey key;             // 要用来加密的密钥
Cipher c = Cipher.getInstance("Blowfish"); // 用来执行加密的对象
c.init(Cipher.ENCRYPT_MODE, key); // 使用密钥来初始化它
SealedObject so = new SealedObject(o, c); // 创建已被封装的对象

// 原始对象 o 经过加密后封装在对象 so 中,
// 现在它可以被序列化并传送
// 这里是接收者解密原始对象的方法
Object original = so.getObject(key); // 必须使用相同的 SecretKey
```

## 各式各样的平台特色

以下的章节详述了 Java 平台重要但繁杂的特色, 包括了特性 (property)、参数选择 (preferences)、进程 (process) 以及管理 (management) 和安装 (instrumentation)。

### 特性

`java.util.Properties` 是 `java.util.Hashtable` 的子类, 是原有的 `collection` 类, 比 Java 1.2 中引入的 `Collections` API 还早。`Properties` 对象会管理字符串键和字符串值之间的映射并定义了一些 `method`, 能把这些映射写入纯文本文件或 (在 Java 5.0 中) XML 文件并读取。这让 `Properties` 类非常适合用于配置以及用户参数选择文件。`Properties` 类也被用于由 `System.getProperty` 返回的系统特性:

```
import java.util.*;
import java.io.*;

// 注意: 如果从未被信任的程序代码 (例如 applets) 中进行调用, 则这些系统特
// 性调用中有很多会抛出安全性异常
String homedir = System.getProperty("user.home"); // 取得一个系统特性
Properties sysprops = System.getProperties(); // 取得所有系统特性

// 列出所有已定义的系统特性的名称
for(Enumeration e = sysprops.propertyNames(); e.hasMoreElements();)
    System.out.println(e.nextElement());

sysprops.list(System.out); // 这有个更简单的输出特性方法
// 从配置文件读取特性
Properties options = new Properties(); // 清空特性列表
File configfile = new File(homedir, ".config"); // 配置文件
```

```

try {
    options.load(new FileInputStream(configfile)); // 从文件载入特性
} catch (IOException e) { /* 在这处理异常 */ }

// 查询特性 ("color"), 如果还没定义, 就指定默认值 ("gray")
String color = options.getProperty("color", "gray");

// 将名称为 "color" 的特性设定为值 "green"
options.setProperty("color", "green");

// 将 Properties 对象的内容存储回文件
try {
    options.store(new FileOutputStream(configfile), // 输出流
                  "MyApp Config File");           // 文件头注释文字
} catch (IOException e) { /* 处理异常 */ }

// 在Java 5.0中, 特性可以被写入XML文件或从XML文件读取
try {
    options.storeToXML(new FileOutputStream(configfile), // 输出流
                       "MyApp Config File");           // 注释文字
    options.loadFromXML(new FileInputStream(configfile)); // 将它读回
}
catch(IOException e) { /* 处理异常 */ }
catch(InvalidPropertiesFormatException e) { /* 变形的输出 */ }

```

## 参数选择

Java 1.4引入了Preferences API, 这是特别用来处理用户和整个系统的参数选择, 它在此用途上比Properties更有用。Preferences API是由java.util.prefs包所定义的, 那个包中的主要类是 `Preferences`。你可以用静态方法 `Preferences.userNodeForPackage()` 取得含有用户特有的参数选择的Preferences对象, 并且用 `Preferences.systemNodeForPackage()` 取得含有整个系统参数选择的Preferences对象。这两个method都会接收 `java.lang.Class` 对象作为唯一的自变量, 并返回由那个包中所有类所共享的Preferences对象 (这代表你所使用的参数选择名称在包中必须是唯一的)。一旦你拥有 Preferences 对象, 就可以使用 `get()` method 来查询具名参数选择的字符串值, 或使用该类型所特有的其他 method, 例如 `getInt()`、`getBoolean()` 以及 `getByteArray()`。请注意, 如果要查询参数选择值, 就必须传递默认值给method。如果具有此指定名称的参数选择还没被注册或保存这些参数选择数据的数据库无法被访问, 就会被返回默认值。Preferences的典型用法如下:

```

package com.davidflanagan.editor;
import java.util.prefs.Preferences;

public class TextEditor {
    // 要由参数选择值初始化的字段
    public int width;           // 以栏数计算的屏幕宽度
    public String dictionary;   // 用于拼写检查的字典名称
}

```



```

public void initPrefs() {
    // 取得针对此包的用户和系统参数选择的 Preferences 对象
    Preferences userprefs = Preferences.userNodeForPackage(TextEditor.class);
    Preferences sysprefs = Preferences.systemNodeForPackage(TextEditor.class);

    // 查询参数选择值。请注意，一定要传递默认值
    width = userprefs.getInt("width", 80);
    // 使用系统参数选择作为默认值来查询用户参数选择
    dictionary = userprefs.get("dictionary",
                               sysprefs.get("dictionary",
                                              "default_dictionary"));
}
}

```

除了用来查询参数选择值的 `get()` method 之外，还有用来设定具名参数选择值的对应 `put()` method:

```

// 用户已指示新的参数选择，所以把它储存起来
userprefs.putBoolean("autosave", false);

```

如果你的应用程序想要在程序运行期间被通知到用户或系统参数选择的改变，它可以用 `addPreferenceChangeListener()` 来注册 `PreferenceChangeListener`。`Preferences` 对象可以将它的参数选择的名称与值导出成为 XML 文件，而且可以从这样的 XML 文件读取参数选择。`Preference` 对象是以层次形式存在的，那通常对应到包名称的层次。用于引导这种层次的 `Preferences` 对象的确存在，但普通的应用程序通常不会用到。

## 进程

在本章的前面，我们看过在同一个 Java 解释器中创建并操作多个运行中的线程有多容易。Java 也有个 `java.lang.Process` 类，它代表了执行于解释器外的操作系统进程。Java 程序可以用流与外部进程通信，就像与运行于网络上的其他计算机中的服务器通信一样。使用 `Process` 一定会有平台依赖性，而且也不便于移植，但它的确很有用。

```

// 通过查询配置文件中所要执行的命令名称来最大化可移植性
java.util.Properties config;
String cmd = config.getProperty("sysloadcmd");
if (cmd != null) {
    // 执行此命令。进程 p 代表执行中的命令
    Process p = Runtime.getRuntime().exec(cmd);
    InputStream pin = p.getInputStream();
    InputStreamReader cin = new InputStreamReader(pin);
    BufferedReader in = new BufferedReader(cin);
    String load = in.readLine();
    in.close();

    // 启动此命令
    // 从中读取字节
    // 将它们转换为字符
    // 读取字符行
    // 取得命令输出
    // 关闭流
}

```

在 Java 5.0 中, `java.lang.ProcessBuilder` 类提供了更具灵活性的方式来启动新进程, 而不是用 `Runtime.exec()` method。 `ProcessBuilder` 使得通过 `Map` 来控制环境变量成为可能, 并让工作目录的设定变得简单。它也有个选项, 可将它所启动的进程的标准错误流自动转到标准输出流, 这让读取 `Process` 的所有输出变得容易。

```
import java.util.Map;
import java.io.*

public class JavaShell {
    public static void main(String[] args) {
        // 我们使用这来启动命令
        ProcessBuilder launcher = new ProcessBuilder();
        // 我们继承来的环境变量。我们可以修改以下这些
        Map<String,String> environment = launcher.environment();
        // 我们的进程会合并错误流与标准输出流
        launcher.redirectErrorStream(true);
        // 读取用户的输入
        BufferedReader console =
            new BufferedReader(new InputStreamReader(System.in));

        while(true) {
            try {
                System.out.print("> ");           // 显示提示
                System.out.flush();               // 强迫显示
                String command = console.readLine(); // 读取输入

                if (command.equals("exit")) return; // exit 命令

                else if (command.startsWith("cd ")) { // 改变目录
                    launcher.directory(new File(command.substring(3)));
                }
                else if (command.startsWith("set ")) { // 设定环境变量
                    command = command.substring(4);
                    int pos = command.indexOf('=');
                    String name = command.substring(0,pos).trim();
                    String var = command.substring(pos+1).trim();
                    environment.put(name, var);
                }

                else { // 否则它是个要启动的进程
                    // 将命令拆散成为个别标记
                    String[] words = command.split(" ");
                    launcher.command(words); // 设定命令
                    Process p = launcher.start(); // 并启动新的进程

                    // 现在从进程读取并显示输出,
                    // 直到不再有输出可读取
                    BufferedReader output = new BufferedReader(
                        new InputStreamReader(p.getInputStream()));
                    String line;
                    while((line = output.readLine()) != null)
                        System.out.println(line);
                }
            }
        }
    }
}
```

```

        // 此进程现在应该完成了，但要等待以确定
        p.waitFor();
    }
}
catch(Exception e) {
    System.out.println(e);
}
}
}
}

```

## 管理与安装

Java 5.0 包含了运行中应用程序的远程监控和管理的强大 JMX API。完整的 javax.management API 超出了本书的范围。其包含的 java.lang.management 包是针对 Java 虚拟机本身的监控和管理的 JMX 应用；java.lang.instrument 是另一个 Java 5.0 包，它允许定义可被用来处理运行中的 JVM 的“代理程序”。在提供支持的 VM 中，java.lang.instrument 可以被用来在类文件被加载时复定义，例如，增加采样或涵盖测试程序代码。类复定义超出了本章的范围，但以下程序代码使用了 Java 5.0 中新出现的安装和管理特性，来判断 Java 程序的资源使用状况。此范例也说明了 Runtime.addShutdownHook() method，它注册了在 VM 开始关机时所运行的程序代码。

```

import java.lang.instrument.*;
import java.lang.management.*;
import java.util.List;
import java.io.*;

public class ResourceUsageAgent {
    // Java agent 类定义了一个 premain() method 以在 main() 之前运行
    public static void premain(final String args, final Instrumentation
inst) {
        // 此代理程序只是注册关机 hook 以在 VM 退出时运行
        Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
                // 这段代码会在 VM 退出时运行
                try {
                    // 决定要把输出送到哪里
                    PrintWriter out;
                    if (args != null && args.length() > 0)
                        out = new PrintWriter(new FileWriter(args));
                    else
                        out = new PrintWriter(System.err);

                    // 使用 java.lang.management 来查询尖峰线程使用状况
                    ThreadMXBean tb = ManagementFactory.getThreadMXBean();
                    out.printf("Current thread count: %d\n",
                        tb.getThreadCount());
                } catch (Exception e) {
                    // 忽略异常
                }
            }
        });
    }
}

```



```

        out.printf("Peak thread count: %d\n",
            tb.getPeakThreadCount());

        // 使用 java.lang.management 来查询尖峰内存使用状况
        List<MemoryPoolMXBean> pools =
            ManagementFactory.getMemoryPoolMXBeans();
        for(MemoryPoolMXBean pool: pools) {
            MemoryUsage peak = pool.getPeakUsage();
            out.printf("Peak %s memory used: %d\n",
                pool.getName(), peak.getUsed());
            out.printf("Peak %s memory reserved: %d\n",
                pool.getName(), peak.getCommitted());
        }

        // 使用传递给 premain() 的 Instrumentation 对象来取得
        // 所有已被载入的类的列表
        Class[] loaded = inst.getAllLoadedClasses();
        out.println("Loaded classes:");
        for(Class c : loaded) out.println(c.getName());

        out.close(); // 关闭并清空输出流
    }

    catch(Throwable t) {
        // shutdown hook 中的异常会被忽略, 所以我们
        // 不会显式地将其列出
        System.err.println("Exception in agent: " + t);
    }
}
});
}
}

```

如果要使用这个代理程序监控Java程序的资源使用状况,你必须先正常地编译此类。接着,把产生出来的类文件和指定了包含 `premain()` method 的清单存储在 JAR 文件中。创建包含这行的清单文件:

```
Premain-Class: ResourceUsageAgent
```

使用像这样的命令来创建 JAR 文件:

```
% jar cmf manifest agent.jar ResourceUsageAgent*.class
```

最后,如果要使用代理程序,就在Java解释器中加上 `-javaagent` 标记来赋值给 JAR 文件和代理程序自变量:

```
% java -javaagent:agent.jar=/tmp/usage.info my.java.Program
```



# Java 的安全性

Java程序可以从各种资源中动态地装载Java类，其中包括不可信的资源，例如通过不安全网络可以访问到的网站。创建并处理这种自由程序代码的能力是Java极大的长处和特性之一。但是，为了让程序能顺利运行，Java非常重视能让不可信的程序代码安全地运行的安全结构，而不必担心对主机系统造成损害。

在Java中对安全系统的需求可由applet看出——applet是个小型的应用程序，被设计来内嵌在网页中（注1）。当用户访问含有applet的网页时（使用支持Java的web浏览器），web浏览器会下载定义了applet的Java类文件并加以运行。如果没有安全系统，applet可能会通过删除文件、安装病毒、窃取机密信息等来在用户的系统上造成混乱。如果applet再巧妙些，就可以利用用户的系统来伪造E-mail、产生垃圾邮件或尝试入侵其他系统。

Java对这种危险程序代码的主要防范方法就是访问控制（access control）：不安全的程序代码就不具有对特定的敏感核心Java API的访问权限。例如，不安全的applet通常不会被允许读取、写入或删除主机系统上的文件，或连接到网络上下载它的web服务器之外的任何计算机。本章会说明Java的访问控制结构以及一些其他方面的Java安全系统。

---

注1： applet在《Java Foundation Classes in a Nutshell》（O'Reilly）中有说明，并未涵盖于本书中，但它们在这仍能作为良好的范例。

## 安全风险

Java从基础上的设计就考虑到安全性,这让它比许多其他已有的系统平台有更大的优势。不过,没有系统能保证100%的安全性,Java也不例外。

Java安全结构是由安全专家设计的,并已被许多其他安全专家研究调查过。其一致的结论就是此结构本身相当坚固而强壮,理论上没有任何安全漏洞(至少还没有被发现)。但是,此安全结构的实现是另一回事,在特定的Java实现中,安全漏洞被找到并修补已有很长的时间了。例如,在1999年4月,Sun在Java 1.1中的类验证器的实现就被发现存在缺陷。针对Java 1.1.6和1.1.7的补丁程序被发布,问题在Java 1.1.8中已解决。在1999年8月,Microsoft的Java Virtual Machine中出现了严重的缺陷。Microsoft修正了问题,并且不再随着它们最新版本的web浏览器发布它们的JVM。

Java VM实现中的安全缺陷很可能会继续被发现(并修补)。尽管如此,Java仍或许是目前最安全的平台。对于恶意Java程序代码利用安全漏洞的个案报道并不多。实际上,Java平台显露出足够的安全性,尤其是与一些不安全且受病毒困扰的平台作比较时。

## JAVA VM 安全性与类文件验证

最底层的Java安全结构牵涉到Java Virtual Machine的设计以及它所执行的字节码。Java VM不允许任何形式的对底层系统个别内存地址的直接访问,这避免了Java程序代码干扰原有的硬件和操作系统。这些在VM上的故意限制也反映在Java语言本身,它不支持指针和指针运算。此语言不允许整数被转化为对象引用,也不允许对象引用被转化为整数,而且没有任何方法可以取得对象在内存中的地址。在没有这些能力的情况下,恶意程序代码毫无立足之地。

除了Virtual Machine指令集的安全设计之外,每当加载不可信的类时,VM就会进行所谓字节码验证(byte-code verification)的过程。此过程确保类的字节码(以及它们的操作数)都是有效的;程序代码绝不会对VM堆栈造成下溢(underflow)或上溢(overflow);局部变量在被初始化之前不会被使用;字段、method以及类访问控制修饰符都会被考虑等。验证步骤是被设计来避免VM运行有问题的程序代码,那些程序代码可能会使VM崩溃或让VM进入未被定义或未被测试的状态,使VM易于被恶意程序代码攻击。字节码验证是对精心编写带有恶意的Java字节码和未被信任的Java编译器的防范,未被信任的Java编译器可能会输出无效的字节码。



## 验证与加密

`java.security`包（及其子包）提供了用于认证的类和接口。就如第五章所描述的，这个安全结构能让Java程序代码创建并验证消息摘要和数字签名。这些技术可以确保所有数据（例如Java类文件）都是可信任的：它来自于真正发出数据的人，而且数据在传输过程中没有被不小心或恶意地修改。

Java Cryptography Extension（或 JCE）包含了 `javax.crypto` 包及其子包，这些包定义了用于数据加密和解密的类。这是对许多应用程序都很重要的安全相关特性，但与避免不可信程序破坏主机系统的基本问题没有直接相关，所以不在本章中讨论。

## 访问控制

就如我们在本章开头所提到的，Java安全结构的中心是访问控制：不可信的程序代码绝不可以被授予对敏感Java API部分的访问权限，若这样会让它能做有恶意的行为。就如我们要在下面的章节讨论的，Java访问控制模型在Java 1.0和Java 1.2之间有重大的发展。从那时开始，访问控制模型就已变得相当稳定，从Java 1.2之后就很少有重大的改变。下个章节提供了Java安全性从Java 1.0到Java 1.2的简短发展历史，它展示了安全模型在最近的主要改变。

### Java 1.0：沙箱

在第一版的Java中，所有安装在本地系统上的Java程序代码都是默认为可信的。但是，所有通过网络下载的程序代码都是不可信的，并且是在被开玩笑称为“沙箱(sandbox)”的受限环境中运行。沙箱的访问控制策略是由当前安装的`java.lang.SecurityManager`对象所定义。当系统程序代码要执行受限制的操作时（例如读取本地文件系统的文件），它会先调用当前安装的`SecurityManager`对象的适当method（例如`checkRead()`）。如果不可信的程序代码正在运行，`SecurityManager`就会抛出`SecurityException`，以避免受限制的操作发生。

最常见的`SecurityManager`类的用户是支持Java的web浏览器，它安装了`SecurityManager`对象来允许applet在不损害主机系统的情况下运行。当然，安全策略的细节是web浏览器的实现细节，但applet通常会被以下几种方式限制：

- applet不可以读取、写入、重命名或删除文件。它不能查询文件的长度、修改日期甚至检查指定文件是否存在。同样地，applet不能创建、列出或删除目录。

- 除了下载 applet 的计算机之外, applet 不能连接到其他计算机或接受来自其他计算机的连接。它不能使用任何特权端口 (亦即端口 1024 以下 (包含 1024) 的端口)。
- applet 不能执行系统层次的功能, 例如加载原生类库、产生新的进程或结束 Java 解释器。除了它本身所创建的之外, applet 不能操作任何线程或线程组。在 Java 1.1 和之后的版本中, 除了随着 applet 一起下载的之外, applet 不能使用 Java Reflection API 来取得关于类的非公共成员的信息。
- applet 不能访问与图形和 GUI 相关的功能。它不能初始化打印工作、访问系统的剪贴板或事件队列。此外, 所有由 applet 创建的窗口通常都会显示明显的可见指示, 标示它们是“不安全的”, 以避免 applet 使用某些其他应用程序的外观来进行欺骗。
- applet 不能读取某些系统特性, 尤其是 `user.home` 和 `user.dir` 特性, 它们定义了用户的根目录和当前工作目录。
- applet 不能通过注册新的 `SecurityManager` 对象来绕过这些安全限制。

## 沙箱的运作方式

假设 applet (或运行于沙箱中的一些其他未被信任的程序代码) 试图通过传递 `/etc/passwd` 这个文件名给 `FileInputStream()` 来读取文件 `/etc/passwd` 的内容。编写 `FileInputStream` 类的程序员察觉到此类提供了对系统资源 (文件) 的访问, 所以此类的使用因而受访问控制的支配。基于此原因, 他们编写了 `FileInputStream()` 构造函数来使用 `SecurityManager` 类。

每当 `FileInputStream()` 被调用时, 它会检查 `SecurityManager` 对象是否已安装。如果安装了, 构造函数就会调用 `SecurityManager` 对象的 `checkRead()` method 将文件名 (在这个例子中是 `/etc/passwd`) 当成唯一的自变量来传递。`checkRead()` method 没有返回值, 它不是正常返回就是抛出 `SecurityException`。如果此 method 返回, 那么 `FileInputStream()` 构造函数就只执行初始化所需的操作并返回; 否则, 它会允许 `SecurityException` 传播至调用者。当这个情况发生时, `FileInputStream` 对象就不会被创建, 而 applet 无法取得对 `/etc/passwd` 文件的访问权。

## Java 1.1: 数字签名类

Java 1.1 保留了 Java 1.0 的沙箱模型, 但增加了 `java.security` 包及其数字签名功能。通过使用这些功能, Java 类可以用数字签名并验证。因此, web 浏览器和其他的 Java 安装可以被设定为包括可信实体的有效数字签名的可信的下载程序代码。这样的程序代码被当成是安装在本地的, 所以它被给予对 Java API 的完全访问权限。在这个版本中, `javakey` 程序管理了以数字方式签名的 Java 程序代码的 JAR 文件。虽然 Java 1.1 增加了

重要的功能来信任以数字方式签名的程序代码（否则程序代码就不会被信任），但它仍坚持基本的沙箱模型：被信任的程序代码能取得所有访问权，而未被信任的程序代码只取得非常受限制的访问权。

## Java 1.2：权限与策略

Java 1.2 把新的访问控制特性引入了 Java 安全结构中，这些特性是由 `java.security` 包中的类所实现。`Policy` 类是最重要的之一：它定义了 Java 安全策略。`Policy` 对象将 `CodeSource` 对象映射至相关联的 `Permission` 对象的集合。`CodeSource` 对象代表一段 Java 程序代码的源代码，它包括了类文件的 URL（可以是在本地的文件）和将数字签名应用至类文件的实体列表。`Permission` 对象与 `Policy` 中的 `CodeSource` 相结合，定义了会被授予来自指定源代码的程序代码的权限。各种 Java API 包括了代表不同类型权限的 `Permission` 子类，包括了 `java.lang.RuntimePermission`、`java.io.FilePermission` 以及 `java.net.SocketPermission` 等。

在此访问控制模型下，`SecurityManager` 类仍然是主要类，访问控制请求仍会通过调用 `SecurityManager` 的 `method` 来完成。但是，默认的 `SecurityManager` 实现将大部分的请求委派至 `AccessController` 类，它会以 `Permission` 和 `Policy` 结构为基础来作出访问决定。

Java 1.2 访问控制结构有几个重要特性：

- 来自不同出处的程序代码可以被给予不同的权限。换言之，此结构支持各种信任层次，甚至于在本地安装的程序代码也可以被当成未被信任或部分未被信任。在此结构下，只有系统类和标准扩展包能被当成完全信任来执行。
- 不再需要定义 `SecurityManager` 的自定义子类来定义安全策略。策略可以由系统管理员通过编辑文本文件或使用 `policytool` 程序来配置，这会在第八章中说明。
- 此结构未被限制于 `SecurityManager` 类中固定的访问控制 `method`。`Permission` 子类可以轻易地被定义来管理对系统资源的访问（例如，通过包含原生程序代码的标准扩展包来接触系统资源）。

## 权限与策略的运作方式

我们回到 applet 的范例，此 applet 尝试创建 `FileInputStream` 来读取文件 `/etc/passwd`。在 Java 1.2 及之后的版本中，`FileInputStream()` 构造函数的行为与 Java 1.0 和 Java 1.1 中的完全一样：它会查看 `SecurityManager` 是否已被安装，如果安装了，就调用它的 `checkRead()` `method` 来传递要被读取的文件名。



在 Java 1.2 中有改变的是 `checkRead()` method 的默认行为。除非程序已用它自己的安全管理员来取代默认的安全管理员,否则默认的实现会创建 `FilePermission` 对象来代表所需的访问。此 `FilePermission` 对象有个 “/etc/passwd” 目标 (target) 并且有个 “读取” 动作 (action)。`checkRead()` method 会把这个 `FilePermission` 对象传递给 `java.security.AccessController` 类的静态 `checkPermission()` method。

在 Java 1.2 中进行真正的访问控制工作的是 `AccessController` 和它的 `checkPermission()` method。此 method 会判定每个进行调用 method 的 `CodeSource`, 并使用当前的 `Policy` 对象来决定与其相关联的 `Permission` 对象。有了这些信息, `AccessController` 就可以决定对 `/etc/passwd` 文件的访问是否应该被允许。

`Permission` 类同时代表了由 `Policy` 所授予的权限以及由像 `FileInputStream()` 这种构造函数所请求的权限。当请求权限时, Java 通常会用到非常明确的目标 (像是 “/etc/passwd”) 的 `FilePermission` (或其他的 `Permission` 子类)。但是, 当授予权限时, `Policy` 通常会以通配符目标代表许多文件 (例如 “/etc/\*”) 来使用 `FilePermission` 对象。`Permission` 子类 (例如 `FilePermission`) 的关键特性之一, 就是它定义了 `implies()` method, 可以判断拥有读取 “/etc/\*” 的权限是否意味着拥有读取 “/etc/passwd” 的权限。

## 针对所有人的安全性

程序员、系统管理员以及终端用户都有不同的安全性考虑, 因此, 在 Java 安全结构中会扮演不同的角色。

## 针对系统程序员的安全性

系统程序员是定义新 Java API 的人, 这些 API 允许对敏感系统资源进行访问。这些程序员通常会运用对系统具有未保护的访问的原生 method。它们必须使用 Java 访问控制结构来避免未被信任的程序代码执行那些原生 method。为了要实现这一点, 系统程序员必须在他们的程序代码中小心地在适当地方插入 `SecurityManager` 调用。系统程序员可能会选择使用已存在的 `Permission` 子类来管理对 API 所能接触的系统资源的访问, 或是可能会决定要定义 `Permission` 的特定子类。

系统程序员担负了极大的安全性重任: 如果在程序代码中没有执行适当的访问控制检查, 就可能会危及整个 Java 平台的安全性。相关的细节很复杂, 已超越了本书的范围。但是, 幸好在 Java 中与原生 method 有关的系统程序设计相当少。我们几乎全都是应用程序员, 可以只依赖已存在的 API。

## 针对应用程序员的安全性

使用了核心 Java API 和标准扩展包但没有定义新扩展包或编写原生 method 的程序员，可以只依赖创建那些 API 的系统程序员在安全性上的努力。换言之，我们大部分的 Java 程序员可以只是使用 Java API，而无需担心将安全性漏洞引进 Java 平台。

事实上，应用程序员很少必须使用访问控制结构。如果你正在编写可能会被当成未被信任的程序代码的 Java 程序代码，就应该会察觉到由典型的安全策略加在未被信任的程序代码上的限制。请记住，有些 method（例如读取与写入文件）可能会抛出 `SecurityException` 对象，但不要觉得你必须自己编写程序代码来捕获这些异常。通常，对 `SecurityException` 的适当响应是允许它不被捕获而被传播出去，以让它终止应用程序。

有时，身为一个应用程序员，你会想编写个应用程序，它可以加载未被信任的类并通过访问控制检查来加以运行。如果要在 Java 1.2 及之后的版本中完成，就必须先安装安全管理器：

```
System.setSecurityManager(new SecurityManager());
```

接着使用 `java.net.URLClassLoader` 来载入未被信任的类。`URLClassLoader` 会指定一组默认的安全权限给它加载的类，但在某些情况下，你可能会想要修改由 `Policy` 和 `PermissionCollection` 类为被加载程序代码授予的权限。

## 针对系统管理员的安全性

在 Java 1.2 和之后的版本中，系统管理员负责定义他们所维护之计算机的默认安全策略。默认的策略被存储在 Java 安装过程中的 `lib/security/java.policy` 文件中。系统管理员可以手动编辑这个文本文件，或使用来自 Sun 的 `policytool` 程序以可视化方式来编辑文件。`policytool` 是较好的定义策略的方式，所以底层策略文件的语法不在本书中说明。

默认的 `java.policy` 文件定义了与 Java 1.0 和 Java 1.1 非常类似的策略：系统类和已安装的扩展包会被完全信任，而其他的程序代码都是未被信任的，只能有一些简单的权限。例如，在一些组织中，将额外的权限授予从安全的内部网络下载的程序代码是适当的。

为了定义有效的安全策略，系统管理员必须了解 Java 平台的各种 `Permission` 子类、它们所支持的目标和行为名称以及授予任一个特定权限的安全实现。这些主题在标题为“Permissions in the Java 2 Standard Edition Development Kit (JDK)”的文件中说明得很详细，这份文件可以在 <http://java.sun.com/j2se/1.5.0/docs/guide/security/permissions.html> 在线取得。

## 针对终端用户的安全性

大部分的终端用户都完全没有考虑到安全性：他们的Java程序应该只能在他们不介入的安全方式下运行。但是，有些具有经验的终端用户可能会想要定义他们自己的安全策略。终端用户可以通过自己运行*policytool*来定义个人策略文件，此文件扩大了系统策略。默认的个人策略是存储于用户根目录中名为*.java.policy*的文件。根据默认，Java会加载此策略文件并使用它来扩大系统策略文件。

在Java 1.2及之后的版本中，当启动Java解释器时，用户可以指定额外的策略文件来使用。如果要这么做，需使用-D选项来定义*java.security.policy*属性。例如：

```
C:\> java -Djava.security.policy=policyfile UntrustedApp
```

该行在使用由文件或URL策略文件所指定的策略来扩大默认的系统策略后运行UntrustedApp类。如果是要取代系统和用户策略而不是加以扩大，可以在属性说明中使用双等号：

```
C:\> java -Djava.security.policy==policyfile UntrustedApp
```

但请注意，只有在SecurityManager已安装的情况下，指定策略文件才会有用。如果用户不信任应用程序，那么他可能不会相信那个应用程序会自动安装它自己的安全管理员。在这样的情况下，他可以定义*java.security.manager*系统属性：

```
C:\> java -Djava.security.manager -Djava.security.policy=policyfile \ UntrustedApp
```

此属性的值并不重要，只要把它定义得足以告诉Java解释器要自动安装默认的SecurityManager对象，以规范应用程序遵守系统、用户和*java.security.policy*策略文件中所描述的访问控制策略即可。

## Permission 类

表6-1列出了一些由核心Java平台定义的重要Permission子类，并归纳了它们所代表的权限。关于这些权限类的完整列表/详细说明、它们的目标和行为目录、method列表以及它们所需的权限，请参阅<http://java.sun.com/j2se/1.5.0/docs/guide/security/permissions.html>（此文件是标准说明文件集的一部分，可以与JDK一起下载）。



表 6-1: Java Permission 类

Permission 类	说明
<code>java.security.AllPermission</code>	此特殊权限类的 instance 包含了所有其他的权限
<code>javax.sound.sampled.AudioPermission</code>	控制播放与记录声音的能力
<code>javax.security.auth.AuthPermission</code>	控制对 <code>javax.security.auth</code> 及其子包中认证 method 进行访问的能力
<code>java.awt.AWTPermission</code>	控制对 <code>java.awt</code> 及其子包中敏感 method 的访问
<code>java.io.FilePermission</code>	管理对文件系统的访问
<code>java.util.logging.LoggingPermission</code>	控制程序修改登录配置 ( <code>logging configuration</code> ) 的能力
<code>java.net.NetPermission</code>	管理对与网络连接相关数据 (例如流处理程序与 HTTP 认证) 的访问。请参阅 <code>java.net.SocketPermission</code>
<code>java.util.PropertyPermission</code>	管理对系统属性的访问
<code>java.lang.reflect.ReflectPermission</code>	管理通过 <code>java.lang.reflect</code> 包对通常无法被访问的类和类成员的访问
<code>java.lang.RuntimePermission</code>	管理对一些 method 和数据的访问。有许多受控制的 method 是由 <code>java.lang.System</code> 和 <code>java.lang.Runtime</code> 所定义
<code>java.security.SecurityPermission</code>	管理对各种与安全性相关的 method 的访问
<code>java.io.SerializablePermission</code>	管理对与序列化相关的 method 的访问
<code>java.net.SocketPermission</code>	管理对网络的访问
<code>java.sql.SQLPermission</code>	管理在 <code>java.sql JDBC API</code> 中指定登录流的能力



# 程序设计与 文档规范

本章说明了一些重要且有用的 Java 程序设计与文档规范。其中包含：

- 一般命名与大小写惯例
- 可移植性提示与惯例
- Javadoc 说明文件注释语法与惯例
- JavaBeans 惯例

在此所说明的惯例都不具强制性。但是，遵循这些惯例能让你的程序代码更易阅读和维护、具有可移植性并能自我注释（self-documenting）。

## 命名与大小写惯例

以下广为采用的命名惯例适用于 Java 中的包、类型、method、字段以及常量。因为这些惯例几乎全世界都遵循，而且因为它们会影响到你定义的类的公共 API，所以应该要小心谨慎地遵守：

### 包

为了确保你的公共可见的包名是唯一的，可以在它们前面加上你经过颠倒的 Internet 域名（例如，com.davidflanigan.utils）。所有的包名都应该是小写。分布于一切完备的 JAR 文件中、由应用程序在内部使用的程序代码包不是公共可见的，所以不必遵循此惯例。在这样的情况下，使用应用程序名称作为包名或包前缀是很常见的。

## 类型

类型名称应该以大写字母开头，而且可以用大小写混合的方式（例如String）。如果类名是由多个单词所组成，那么其中的每个单词都应该以大写字母开头（例如，StringBuffer）。如果类型名称或类型名称中的其中一个单词是字母缩写，则此字母缩写就应该全部用大写字母编写（例如URL、HTMLParser）。

因为类和枚举类型是被设计来代表对象，所以你应该选择名词形式的类名（例如Thread、Teapot、FormatConverter）。

当interface被用来提供关于实现此interface类的额外信息时，选择具有形容词形式的interface名称是很常见的（例如Runnable、Cloneable、Serializable）。annotation类型也常是用这种方式命名。当interface的运行较像是抽象父类时，就使用名词形式的名称（例如Document、FileNameMap、Collection）。

## method

method名称一定应以小写字母开头。如果名称包含了多个单词，那么在第一个单词之后的每个单词都应以大写字母开头（例如insert()、insertObject()、insertObjectAt()）。method名称通常都是经过精心设计的，所以一般它们的第一个单词是动词。method名称可以用需要的长度让它们的用途清楚，但要尽可能选择简洁的名称。

## 字段与常量

非常量字段的名称遵循着与method名称相同的大小写惯例。如果字段是static final 常量，就应该使用大写来编写。如果常量的名称包含了多个单词，那么这些单词就应该以下划线隔开（例如MAX\_VALUE）。字段名称应该选择最能说明字段或其取值的含义的。

由枚举类型定义的常量通常也会都用大写字母表示。但是，因为其他的程序设计语言对枚举值使用小写或大小写混合，所以此惯例并没有像在类或interface的static final 字段中使用大写字母的惯例一样具有权威。

## 参数

method参数遵循与非常量字段相同的大小写惯例。method参数的名称出现在method的说明文件中，所以你应该选择能让参数的用途尽可能清楚的名称。要试着让参数名称为一个单词并以一致的方式来使用它们。例如，如果WidgetProcessor类定义了许多会接受Widget对象作为第一参数的method，就在每个method中把这个参数命名为widget 甚至是w。

## 局部变量

局部变量名称是实现细节，在类的外部绝不会看到。不过，选择好的名称能让你的



程序代码较易于阅读、理解以及维护。变量的命名通常会遵循与 `method` 和字段相同的惯例。

除了针对特殊名称类型的惯例之外，还有一些与名称中所使用的字符有关的惯例，应该在你的名称中使用。Java 允许标识符中有 `$` 字符，但依惯例，此用法是留给由源代码处理程序所产生的合成名称（例如，Java 编译器用它来让内部类运行）。此外，Java 允许对名称使用来自整个 Unicode 字符集的字母和数字字符（alphanumeric character）。虽然这对于非英语系（non-English-speaking）的程序员很方便，但 Unicode 字符的使用应该要被限制在局部变量、私有 `method` 和字段以及不属于类的公共 API 中的其他名称。

## 可移植性惯例和纯 Java 规则

Sun 的格言，或者说是它对于 Java 的核心价值论点就是 “Write once, run anywhere”。Java 使得具有可移植性程序的编写变得容易，但 Java 程序不能自动在所有的 Java 平台上顺利运行。以下提示能帮助避免出现可移植性方面的问题。像此处所列出的可移植性规则，是 Sun 现在已废止的 “100% Pure Java” 认证项目与品牌活动的焦点。

### *native method*

具有可移植性的 Java 程序代码可以使用核心 Java API 中的所有 `method`，其中包括被实现为原生（native）`method` 的那些 `method`。但是，具有可移植性的程序代码绝不可以定义它自己的 `native method`。就其本质来看，`native method` 必须被移植到各个新平台，所以它们直接破坏了 Java 的 “Write once, run anywhere” 承诺。

### *Runtime.exec() method*

调用 `Runtime.exec()` `method` 来产生一个进程并执行原生系统上的外部命令，在具有可移植性的程序代码中是很少被允许的。这是因为要被执行的原生 OS 命令无法保证在所有的平台上都存在或都以同样的方式执行。唯一能合法使用 `Runtime.exec()` 的时机，就是在用户被允许指定所要执行的命令时。这可以通过在运行时输入命令或通过配置在配置文件或参数选择对话框中指定。

### *System.getenv() method*

使用 `System.getenv()` 是不具有可移植性的。此 `method` 之前已废止，但在 Java 5.0 中被再次引入。

### 无说明文件的类

具有可移植性的 Java 程序代码只能使用 Java 平台的有说明文件部分的类和接口。大部分的 Java 实现都会有额外的无说明文件的公共类，那些是实现的一部分，但不是 Java 平台的一部分。没有任何方式能避免程序使用与依赖这些无说明文件的

类，但这么做是不具有可移植性的，因为无法保证这些类在所有的 Java 实现或平台上都存在。

#### *java.awt.peer* 包

*java.awt.peer* 包中的接口是 Java 平台的一部分，但只是对由 AWT 实现的用法做使用说明。直接使用这些接口的应用程序是不具可移植性的。

#### 针对专门实现 (*Implementation-specific*) 的特性

具有可移植性的程序代码绝不可以依赖单独实现所特有的特性。例如，Microsoft 发布了一个 Java 运行时系统的版本，其中包含了一些 *method*，但那些 *method* 并非由 Sun 定义为 Java 平台的一部分。很明显地，任何依赖这种扩展功能的程序会无法被移植到其他的平台。Microsoft 的 Java 平台私有扩展功能造成了 Sun 和 Microsoft 之间的法律纠纷，最终使得 Microsoft 不再支持 Java。

#### 针对专门实现的缺陷

就和具有可移植性的程序代码绝不可以依赖针对专门实现的特性一样，它也绝不可以依赖针对专门实现的缺陷。如果类或 *method* 的行为与规范所定义的不同，具有可移植性的程序就不可以依赖这个行为，因为那可能在不同的平台上会有差异且最终会被修正。

#### 针对专门实现的行为

有时不同的平台和不同的实现会呈现出不同的行为，依据 Java 的规范来看，那些都是合法的。具有可移植性的程序代码绝不可以依赖任何一个特定的行为。例如，Java 规范没有规定具有相等优先级的线程可共享 CPU，或是长时间运行的线程可以激活另一个具有相同优先级的线程。如果应用程序假设了其中一个特定的行为，那么它就可能不会在所有的平台上都正确地运行。

#### 标准扩展

具有可移植性的程序代码可以依赖针对 Java 平台的标准扩展，但如果这么做，就应该明确地指出它使用的是哪个扩展，并当它在未安装那个扩展的系统上运行时，要有明确地给出的适当错误信息。

#### 完整的程序

所有具有可移植性的 Java 程序都必须完整且独立：除了核心平台和标准扩展类之外，它必须提供所有它要使用的类。

#### 定义系统类

具有可移植性的 Java 程序代码绝不会在任何的系统或标准扩展包中定义类。这样做破坏了那些包的保护界限，并且暴露了在包可见的实现细节。

### 硬编码文件名

具有可移植性的程序不会包含硬编码的 (hardcoded) 文件或目录名称。这是因为不同的平台会有差异极大的文件系统组织方式, 并使用了不同的目录分隔字符。如果你需要处理文件或目录, 就要让用户指定文件名或至少要指定可找到此文件的基目录。此声明可以在运行时在配置文件中或作为程序的命令行自变量来完成。当把文件或目录名称与目录名称合并时, 要使用 `File()` 构造函数或 `File.separator` 常量。

### 换行字符

不同的系统使用了不同的字符或字符序列作为换行字符 (line separator)。不要把 `\n`、`\r` 或 `\r\n` 硬编码为程序的换行字符, 而是要用 `PrintStream` 或 `PrintWriter` 的 `println()` method, 它会自动以适合于此平台的换行字符来结束一行或是使用 `line.separator` 系统属性值。在 Java 5.0 及之后的版本中, 你也可以对 `java.util.Formatter` 及相关类的 `printf()` 和 `format()` method 使用 `"%n"`。

## Java 说明文件的注释

大部分 Java 程序代码中的普通注释是在说明程序代码的实现细节。相较之下, Java 语言规范定义了所谓 *doc* 注释的特殊注释类型, 用来说明你的程序代码的 API。doc 注释是以 `/**` 开始 (而不是一般的 `/*`) 并以 `*/` 结束的普通多行注释, 它会出现于类型或成员定义的前面, 并包含针对那个类型或成员的说明文件。此说明文件可以包含简单的 HTML 格式化标记和其他提供额外信息的特殊关键字。doc 注释会被编译器忽略, 但它们可以由 *javadoc* 程序被提取出来并转换成在线的 HTML 说明文件 (关于 *javadoc* 的更多信息, 请参阅第八章)。这有个包含了适当 doc 注释的范例类:

```
/**
 * 这个具有不变性的类代表了 <i>复数</i>
 *
 * @作者 David Flanagan
 * @版本 1.0
 */
public class Complex {
    /**
     * 存放此复数的实数部分
     * @参阅 #y
     */
    protected double x;

    /**
     * 存放此复数的虚数部分
     * @参阅 #x
     */
}
```





```
    */
    protected double y;

    /**
     * 创建新的 Complex 对象来代表复数 x+yi
     * @参数 x 复数的实数部分
     * @参数 y 复数的虚数部分
     */
    public Complex(double x, double y) {
        this.x = x;
        this.y = y;
    }

    /**
     * 将两个 Complex 对象相加并产生代表它们总和的第三个对象
     * @参数 c1 Complex 对象
     * @参数 c2 另一个 Complex 对象
     * @返回用来代表 <code>c1</code> 和 <code>c2</code> 总和
     * 的新 Complex 对象
     * @异常 java.lang.NullPointerException
     * 如果其中有个自变量是 <code>null</code>
     */
    public static Complex add(Complex c1, Complex c2) {
        return new Complex(c1.x + c2.x, c1.y + c2.y);
    }
}
```

## doc 注释的结构

doc 注释的主体应该要以关于要被说明的类型或成员的一句话摘要来开始。这句话本身可以显示作为归纳的说明文件，所以它应该被独立编写。在起始句子的后面可以接着任意数量的其他句子和段落，用来更详细地描述类、接口、method 或字段。

在具描述性的段落后面，doc 注释可以包含任意数量的其他段落，其中每个段落都是以特殊的 doc 注释标记开始，例如 @author、@param 或 @returns。这些加上标记的段落提供了关于类、接口、method 或字段的专门信息，可让 javadoc 程序以标准的方式来显示。完整的 doc 注释标记列于接下来的段落中。

在 doc 注释中，具有描述性的信息可以包含简单的 HTML 标注标记，例如用于强调的 <I>、用于类、method 以及字段名称的 <code> 以及用于多行程序代码范例的 <pre>。它也可以包含 <p> 标记来将语句拆为分隔的段落以及用 <ul>、<li> 和相关的标记来显示加上标号的列表和类似的结构。但请记住，你所编写的信息是被内嵌在更大、更复杂的 HTML 文件中。基于此原因，doc 注释不应该包含 HTML 的主要结构标记，例如 <h2> 或 <hr>，因为那可能会干扰到较大文档的结构。

要避免在 doc 注释中使用 <a> 标记来引入超级链接或交叉引用，而是使用特殊的

{@link} doc 注释标记，它与其他的 doc 注释标记不同，可以出现在 doc 注释中的任何地方。就如在接下来的章节中所描述的，{@link} 标记能让你在不知道 *javadoc* 所使用的 HTML 结构惯例和文件名的情况下，指定对其他类、接口、method 以及字段的超级链接。

如果你想在 doc 注释中加上图像，可以把图像文件放在源代码目录的 *doc-files* 子目录中。让图像与类具有相同的名称并加上整数后缀。例如，出现于名称为 *Circle* 的类的 doc 文件中的第二张图片，可以用这个 HTML 标记来定义：

```

```

因为这些 doc 注释是被内嵌在 Java 注释中，所以在处理之前，每一行注释的前置空白和星号都会被剔除。因此，你不必担心星号出现在最后产生的说明文件中，或担心注释的缩排会影响到以 `<pre>` 标记被包含于注释中的程序代码范例的缩排。

## doc 注释标记

*javadoc* 能识别一些专门的标记，都是以 @ 字符开始。这些 doc 注释标记能让你以标准的方式将专门的信息编码到你的注释中，而且它们能让 *javadoc* 选择针对那份信息的适当输出格式。例如，@param 标记让你指定 method 的单一参数名称和意义。*javadoc* 可以提取出这份信息，并使用 HTML 的 `<dl>` 列表、HTML 的 `<table>` 或任何适合的方式来加以显示。

以下的 doc 注释标记能被 *javadoc* 识别。doc 注释通常应该会以此处所列的顺序来使用这些标记：

### @author name

加入包含指定名称的“Author:”项。此标记应该针对每个类或接口定义来使用，但绝不可以用于单独的 method 和字段。如果一个类有多个作者，那么就在相邻的行上使用多个 @author 标记。例如：

```
@author David Flanagan  
@author Paula Ferguson
```

依时间顺序列出作者，原始作者放在第一个。如果没人知道作者，可以使用“unasccribed”。*javadoc* 不会输出作者信息，除非定义了 -author 命令行自变量。

### @version text

插入含有指定文本的“Version:”项。例如：

```
@version 1.32, 08/26/04
```

此标记应该被包含在每个类和接口的 doc 注释中，但不可以用于单独的 method 和

字段。它常与版本控制系统（例如SCCS、RCS或CVS）的版本自动编号功能一起使用。*javadoc* 不会在产生的说明文件中输出版本信息，除非定义了 `-version` 命令行自变量。

`@param parameter-name description`

对当前 *method* 的“Parameters:”段落增加指定的参数及其描述。*method* 或构造函数的 *doc* 注释必须为 *method* 的每个参数包含一个 `@param`。这些标记出现的顺序应该与 *method* 指定的参数的顺序相同。此标记只能用于 *method* 和构造函数的 *doc* 注释中。你应该尽可能使用词组和语句段以让描述保持简短。但是，如果参数需要有详细的说明文件，则此描述可以覆盖数行并包含任意数量的文字。针对源代码形式上的可读性，要考虑使用空格来让这些描述彼此对齐。例如：

```
@param o        要被插入的对象
@param index    所要插入的位置
```

`@return description`

插入含有特定描述的“Returns:”段落。此标记应该出现在 *method* 的每个 *doc* 注释中，除非 *method* 返回 `void` 或它是个构造函数。描述的长度可以依需要决定，但请考虑使用语句段来保持简短。例如：

```
@return <code>true</code> if the insertion is successful, or
        <code>false</code> if the list already contains the specified object.
```

`@exception full-classname description`

加入包含了指定异常名称和描述的“Throws:”项。针对出现在 *throws* 子句中的每个已校验异常，*method* 或构造函数的 *doc* 注释应该包含一个 `@exception` 标记。例如：

```
@exception java.io.FileNotFoundException
           If the specified file could not be found
```

当有些异常是 *method* 的用户想要捕获的异常时，`@exception` 标记可以被选择性地用来说明 *method* 可能抛出的未校验异常（亦即 *RuntimeException* 的子类）。如果 *method* 可以抛出多个异常，那么就在相邻行上使用多个 `@exception` 标记并依字母顺序列出这些异常。此描述可以依所需尽可能短或长，以说明异常的重要性。该标记只能用于 *method* 和构造函数注释。`@throws` 标记是 `@exception` 的同义词。

`@throws full-classname description`

此标记是 `@exception` 的同义词。

`@see reference`

加入含有指定引用的“See Also:”项。此标记可以出现在任何类型的 *doc* 注释中。针对 *reference* 的语法会在本章稍后的“*doc* 注释中的交叉引用”中说明。



### @deprecated *explanation*

此标记说明了接下来的类型或成员已过时，应避免使用。*javadoc*会对说明文件加上显著的“Deprecated”项并包含指定的*explanation*文字。此段文字应说明类或成员何时会过时，如果有可能，还要提供替代的类或成员并包含对其的链接。例如：

```
@deprecated As of Version 3.0, this method is replaced  
            by {@link #setColor}.
```

虽然Java编译器会忽略所有的注释，但它会注意到doc注释中的@deprecated标记。当此标记出现时，编译器会注意到它所产生的类文件中的过时问题。这让它能对其他依赖已过时特性的类提出警告。

### @since *version*

说明类型或成员何时会被加入API。此标记后面应该接着版本号或其他版本规范。例如：

```
@since JNUT 3.0
```

每个针对类型的doc注释都应该包含一个@since标记，而对于在类型的最初版本后加入的任一个成员，在它们的doc注释中都应该要有@since标记。

### @serial *description*

严格说来，类被序列化的方式是它的公共API的一部分。如果你编写了预期会被序列化的类，就应该用@serial和以下所列出的相关标记来说明它的序列化格式。对于任一个属于Serializable类的序列化状态的一部分的字段，@serial应该要出现在doc注释中。对于使用默认序列化机制的类，这表示了所有未被声明为transient的字段，其中包括了被声明为private的字段。*description*应该是字段及它在序列化对象中用途的简短描述。

在Java 1.4中，你也可以在类和包层次上使用@serial标记来指定“序列化格式页”是否应该针对类或包而产生。语法为：

```
@serial include  
@serial exclude
```

### @serialField *name type description*

Serializable类可以通过在名称为serialPersistentFields的字段中声明由ObjectStreamField对象所组成的对象，来定义它的序列化格式。对于这样的类，针对serialPersistentFields的doc注释应该要包含针对数组的每个元素的@serialField标记。每个标记指定了类的序列化状态中特定字段的名称、类型和描述。

### @serialData *description*

Serializable类可以定义一个writeObject() method来写入由默认的序列化

机制所写入的之外的数据。Externalizable 类定义了一个 writeExternal() method, 来负责将对象的完整状态写至序列化流。@serialData 标记应该被使用在针对这些 writeObject() 和 writeExternal() method 的 doc 注释中, 而 description 应该要说明 method 所使用的序列化格式。

## 在线 doc 注释标记

除了前面的标记之外, javadoc 也支持好几种在线标记 (inline tag), 它们可以出现在 HTML 文本在 doc 注释中所能出现的任何地方。因为这些标记会直接出现在 HTML 文本流中, 所以必须要用大括号作为分隔字符来隔开标记文本与 HTML 文本。Java 所支持的在线标记包括了以下几种:

{@link reference}

在 Java 1.2 及之后的版本中, {@link} 标记就像是 @see 标记, 除了它不是在特殊的 “See Also:” 段落中放置对指定引用的链接, 而是在线插入链接。{@link} 标记可以出现在 HTML 文本在 doc 注释中所能出现的任何地方。换言之, 它可以出现在类、接口、method 或字段的初始描述中以及在与 @param、@returns、@exception 和 @deprecated 标记相关联的描述中。{@link} 的 reference 使用了接下来在 “doc 注释中的交叉引用” 中所描述的语法。例如:

```
@param regexp The regular expression to search for. This string
               argument must follow the syntax rules described for
               {@link java.util.regex.Pattern}.
```

{@linkplain reference}

在 Java 1.4 及之后的版本中, {@linkplain} 标记就像是 {@link}, 除了链接的文本是用一般的字型来格式化, 而不是 {@link} 标记所使用的程序代码字型。这在 reference 同时包含链接 feature 以及说明显示于链接中的替代文字的 label 时最有用。关于 reference 自变量的 feature 和 label 部分的讨论, 请参阅 “doc 注释中的交叉引用”。

{@inheritDoc}

当一个 method 覆盖父类中的 method 或实现接口中的 method 时, 你可以省略 doc 注释, javadoc 会自动继承来自于被覆盖或被实现 method 的说明文件。但在 Java 1.4 中, {@inheritDoc} 标记能让你继承单独标记中的文字。此标记也允许你继承及扩大具有描述性的注释文字。如果要继承单独标记, 可以这样:

```
@param index {@inheritDoc}
@return {@inheritDoc}
```

如果要继承整个 doc 注释, 把你自己的文字加在它之前或之后, 可以这样使用标记:

```
This method overrides {@link java.lang.Object#toString}, documented as follows:  
<P>{@inheritDoc}  
<P>This overridden version of the method returns a string of the form...
```

`{@docRoot}`

此在线标记没有接收参数，而且会以对产生的说明文件所在根目录的引用来取代。这对引用外部文件（例如图像或版权声明）的超级链接很有用：

```
  
This is <a href="{@docRoot}/legal.html">Copyrighted</a> material.
```

`{@docRoot}` 是在 Java 1.3 中引入的。

`{@literal text}`

此在线标记会照字面显示文字，它会略过其中所有的 HTML 并忽略可能包含的 *javadoc* 标记。它不会保持留白格式，但在 `<pre>` 标记中留白格式很有用。`{@literal}` 于 Java 5.0 及之后的版本中被提供。

`{@code text}`

此标记就像 `{@literal}` 标记，但会以程序代码字型显示直接量文字。等于是：

```
<code>{@literal text}</code>
```

`{@code}` 于 Java 5.0 及之后的版本中被提供。

`{@value}`

`{@value}` 标记在不具有自变量的情况下，被在线用于 `static final` 字段的 doc 注释中并会被那个字段的常量值所取代。此标记是在 Java 1.4 中引入的，而且只能用于常量字段。

`{@value reference}`

这是 `{@value}` 的变体，包含了对 `static final` 字段的 *reference*，它会被那个字段的常量值所取代。虽然无自变量版本的 `{@value}` 标记是在 Java 1.4 中引入的，但这个版本只有在 Java 5.0 及之后的版本中才被提供。关于 *reference* 的语法，请参阅“doc 注释中的交叉引用”。

## doc 注释中的交叉引用

`@see` 标记以及在线标记 `{@link}`、`{@linkplain}` 和 `{@value}` 都把交叉引用编码成一些其他的说明文件源代码，通常是针对其他类型或成员的说明文件注释。

*reference* 可以采用三种不同的形式。如果它以引号字符开始，它就会被当成书籍名称或某种可打印资源并以此被显示。如果 *reference* 是以 `<` 字符开始，它就会被当成使用了 `<a>` 标记的 HTML 超级链接，而超级链接会被插入输出的说明文件中。此形式的 `@see` 标记可以插入链接到其他的在线文件，例如程序员指南或用户手册。



如果 *reference* 不是被引号括起的字符串或超级链接, 那么它就会被预期为具有以下形式:

*feature label*

在这种情况下, *javadoc* 输出由 *label* 所指定的文字并将它编码为对指定 *feature* 的超级链接。如果 *label* 被省略 (通常会如此), *javadoc* 就使用所指定的 *feature* 的名称来代替。

*feature* 可以引用包、类型或类型成员, 使用的是以下的其中一种形式:

#### *pkgname*

对具名包的调用。例如:

```
@see java.lang.reflect
```

#### *pkgname.typeName*

对由完整包名所指定的类、接口、枚举类型或 *annotation* 类型的引用。例如:

```
@see java.util.List
```

#### *typeName*

在没有加上包名的情况下, 对所指定的类型的引用。例如:

```
@see List
```

*javadoc* 会通过搜索当前包及已导入类的列表来解析具有这个名称的引用。

#### *typeName#methodName*

对指定类型中具名 *method* 或构造函数的引用。例如:

```
@see java.io.InputStream#reset  
@see InputStream#close
```

如果类型是在没有包名的情况下被指定, 它就会以对 *typeName* 所述的方式那样来解析。如果 *method* 被重载或类以相同名称定义了字段, 那么此语法就会含糊不清。

#### *typeName#methodName(paramtypes)*

以参数显式指定的类型对 *method* 或构造函数的引用。这在交叉引用重载的 *method* 时很有用。例如:

```
@see InputStream#read(byte[], int, int)
```

#### *#methodName*

在当前类、接口或包含类、超类、当前类或接口的超接口的其中之一中, 对非重载的 *method* 或构造函数的引用, 使用这种简洁的形式来引用同一个类中的其他 *method*。例如:

```
@see #setBackgroundColor
```

### **#methodname(paramtypes)**

在当前类、接口或包含类、超类的其中之一中对 method 或构造函数的引用。此形式之所以能处理重载的 method, 是因为它显式地列出了 method 参数的类型。例如:

```
@see #setPosition(int, int)
```

### **typename#fieldname**

对指定类中具名字段的引用。例如:

```
@see java.io.BufferedReader#buf
```

如果类型是在没有包名的情况下被指定, 就会以如针对 *typename* 所描述的方式那样来解析。

### **#fieldname**

对当前类型或包含类、超类或当前类型的超接口其中之一中的字段的调用。例如:

```
@see #x
```

## 包的 doc 注释

针对类、接口、method、构造函数以及字段的说明文件注释, 会出现在 Java 源代码中它们所要说明的特性的定义前面。javadoc 也可以读取并显示包的概括性说明文件。由于包被定义在目录中而不是在单一的源代码文件中, 所以 javadoc 会在包含此包的类的源代码的目录中, 寻找名为 *package.html* 的文件, 其中有包的说明文件。

*package.html* 文件应该包含针对包的简单 HTML 说明文件。它也可以包含 @see、@link、@deprecated 以及 @since 标记。由于 *package.html* 不是 Java 源代码的文件, 所以它所包含的说明文件应该是 HTML 且不会是 Java 注释 (亦即它不应该被包括在 */\*\** 和 *\*/* 字符中)。最后, 所有出现在 *package.html* 中的 @see 和 @link 都必须使用完全限定的类名。

除了定义针对每个包的 *package.html* 文件之外, 你也可以通过在那些包的源码树 (source tree) 中定义一个 *overview.html* 文件来对一组包提供高层次的说明文件。当 javadoc 在该源码树上运行时, 就会使用 *overview.html* 作为它所显示的最高层次概观。

## JavaBeans 惯例

JavaBeans 是用来定义可重用模块化软件组件的结构。JavaBeans 声明包含了以下 bean 的定义: “一个具有可重用性的软件组件, 它可以在程序建立工具 (builder tool) 中以可视化的方式来操作”。就如你所看到的, 这是个相当宽松的定义, bean 可以有多种形式。bean 最常见的用处就是用于图形用户界面组件, 例如 java.awt 和 javax.swing 包

的组件。这些在《Java Foundation Classes in a Nutshell》和《Java Swing》中说明，这两本书都是由O'Reilly出版。虽然所有的bean都可以用可视化的方式来操作，但这并不代表每个bean都有自己的可视化表示方式。例如，`javax.sql.RowSet`类（说明于O'Reilly的《Java Enterprise in a Nutshell》中）是JavaBeans组件，它代表由数据库查询所产生的数据。其实对JavaBeans组件的简单性和复杂性上并没有任何限制。最简单的bean通常是基本的图形界面组件，例如`java.awt.Button`对象。但即使复杂的系统，例如具可嵌入性的电子数据表程序包，也可以作为一个单独的bean运行。

JavaBeans组件模型是由`java.beans`、`java.beans.beancontext`包以及一些重要的命名和API惯例所组成，那些是bean和bean操作工具必须遵守的规范。这些惯例不是JavaBeans API本身的一部分，但对bean开发者来说，它们在许多方面都比API本身更为重要。这些惯例有时被称为设计模式（design pattern），它们指定了像method名称和由bean定义的特性访问（property accessor）method的签名这样的内容。如果你所编写的类不是要成为bean，以适合在程序建立工具中做可视化的操作，那么就不必遵循这些惯例。但是，JavaBeans惯例已被广泛使用并理解，你可以通过遵循相关的惯例来改善你的程序代码的可用性和重用性，尤其在property accessor method的命名惯例上更是如此。

我们会在本节的稍后说明这些惯例本身。但是，先浏览JavaBeans模型是十分必要的。

## Bean 基础

任何符合特定基本规则的对象都可以是bean，所有的bean都不必是Bean类的子类。有许多的bean是GUI组件，编写不会出现在屏幕上的“不可见”bean也是相当有可能的而且通常很有用（但是，在完成的应用程序中不会出现在屏幕上的bean并不代表就不能用beanbox工具来做可视化的操作）。

bean可以由它导出的属性（property）、事件（event）以及method来描绘。property是bean的一个内部状态，可用程序来设定并查询，这通常会通过标准的get和set accessor method来进行。

bean会通过产生事件来与它所嵌入的应用程序以及其他bean沟通。JavaBeans API使用与AWT和Swing组件所使用的相同的事件模型。此模型是以`java.util.EventObject`类和`java.util.EventListener`接口为基础，这部分在《Java Foundation Classes in a Nutshell》（O'Reilly）中有详细的说明。简言之，事件模型的运作就像这样：

- bean在定义事件时，会提供add和remove method以针对那个事件的监听对象（listener object）进行注册和注销监听。



- 应用程序如果要在那个类型的事件发生时被通知，可以使用这些 method 来注册相应类型的事件监听对象。
- 当事件发生时，bean 会通过传递描述此事件的事件对象给由事件监听接口所定义的 method，来通知所有已注册的监听程序。

单点传播 (unicast) 事件是相当罕见的事件种类，在这样的情况下只有单一已注册的监听对象。如果尝试注册多于一个的监听程序，单点传播事件的 add 注册 method 就会抛出 TooManyListenersException。

由 bean 导出的 method 只是由 bean 所定义的公共 method，其中不包括取得并设定属性值和注册与注销事件监听程序的那些 method。

除了先前所描述的常规属性之外，JavaBeans API 也支持好几种专门的属性子类型。变址属性 (indexed property) 就是具有数组值的属性，还包括能访问数组元素和整个数组的读取函数 (getter) 与设定函数 (setter) 的 method。约束属性 (bound property) 会在每当属性的值改变时，就发送 PropertyChangeEvent 给任何有兴趣的 PropertyChangeListener 对象。限定属性 (constrained property) 的任何改变都可以被任何感兴趣的监听程序所否决。当 bean 的限定属性的值改变时，bean 必须发送 PropertyChangeEvent 给所有感兴趣的 VetoableChangeListener 对象。如果有任一个对象抛出 PropertyVetoException，则此属性值就不会被更改，而 PropertyVetoException 会被传播回属性的 setter method。

## bean 类

bean 类本身必须遵循以下惯例：

### 类名

在 bean 的类名上没有任何限制。

### 超类

bean 可以扩展任何其他的类。bean 常是 AWT 或 Swing 组件，但这里并没有什么限制。

### 实例化

bean 应该提供不带参数的构造函数，以让 bean 操作工具可以轻易地实例化 bean。

## 属性 (property)

假设 bean 具有遵循这些模式的 accessor method，它定义了类型 T 的属性 p (如果 T 是 boolean 类型的，那么专门形式的 getter method 是被允许的)：

### Getter

```
public T getP()
```

### Boolean getter

```
public boolean isP()
```

### Setter

```
public void setP(T)
```

### Exception

属性 accessor method 可以抛出任何类型的已校验或未校验的异常。

## 变址 (indexed) 属性

变址属性是属于数组类型的属性,它提供了能读取与设置整个数组的 accessor method 以及能读取和设置每个单独数组元素的 method。如果 bean 定义了以下的 accessor method, 就是定义了属于类型  $T[]$  的变址属性  $p$ :

### Array getter

```
public T[] getP()
```

### Element getter

```
public T getP(int)
```

### Array setter

```
public void setP(T[])
```

### Element setter

```
public void setP(int,T)
```

### Exception

变址属性 accessor method 可以抛出任何类型的已校验或未校验的异常。如果所提供的地址超出界限, 它们就应该抛出 `ArrayIndexOutOfBoundsException`。

## 约束 (bound) 属性

约束属性就是在它的值改变时, 它会产生 `PropertyChangeEvent` 的属性。以下是约束属性的惯例:

### Accessor method

约束属性的 getter 和 setter method 遵循与一般属性相同的惯例。

### 监听程序的注册

定义了一或多个约束属特性的 bean 必须定义一对 method 以供监听程序的注册, 这些监听程序会在任意一个约束属性值改变时被通知。那些 method 必须具有这些签名:

```
public void addPropertyChangeListener(PropertyChangeListener)
public void removePropertyChangeListener(PropertyChangeListener)
```

### 具名的属性监听程序的注册

bean 可以选择性地提供额外的 method, 让事件监听程序能针对单独约束属性值的改变来注册。属性的名称会被传递给这些 method 且 method 具有以下签名:

```
public void addPropertyChangeListener(String, PropertyChangeListener)
public void removePropertyChangeListener(String, PropertyChangeListener)
```

### 单属性 (per-property) 监听程序的注册

bean 可以选择性地提供额外的专属于单独属性的事件监听程序注册 method。对于属性 *p*, 这些 method 具有以下签名:

```
public void addPListener(PropertyChangeListener)
public void removePListener(PropertyChangeListener)
```

这种类型的 method 允许用 beanbox 来区分约束属性与非约束属性。

### 通知 (notification)

当约束属性的值改变时, bean 就应该更新它的内部状态以反映此变化, 然后传递 `PropertyChangeEvent` 给每个注册这个 bean 或专门的约束属性的 `PropertyChangeListener` 对象的 `propertyChange()` method。

### 支持

`java.beans.PropertyChangeSupport` 对于实现约束属性是很有帮助的。

## 限定属性

限定属性的任何改变都可以由已注册的监听程序否决。大部分的限定属性也是约束属性。以下是限定属性的惯例:

### Getter

限定属性的 getter method 与一般属性的 getter method 相同。

### Setter

如果属性的改变被否决, 限定属性的 setter method 就会抛出 `PropertyVetoException`。对于属于类型 *T* 的属性 *p*, 签名看起来就像这样:

```
public void setP(T) throws PropertyVetoException
```



### 监听程序的注册

定义了一个或多个限定属性的 bean 必须定义一对 method 以供监听程序注册, 这些监听程序会在任一个限定属性值改变时被通知。那些 method 必须具有这些签名:

```
public void addVetoableChangeListener(VetoableChangeListener)
public void removeVetoableChangeListener(VetoableChangeListener)
```

### 具名的属性监听程序的注册

bean 可以选择性地提供额外的 method, 让事件监听程序能针对单独限定属性值的改变来注册。属性的名称会被传递给这些 method 且 method 具有以下签名:

```
public void addVetoableChangeListener(String, VetoableChangeListener)
public void removeVetoableChangeListener(String, VetoableChangeListener)
```

### 单属性监听程序的注册

bean 可以选择性地提供额外的专属于单独限定属性的监听程序注册 method。对于属性 *p*, 这些 method 具有以下签名:

```
public void addPListener(VetoableChangeListener)
public void removePListener(VetoableChangeListener)
```

### 通知

当限定属性的 setter method 被调用时, bean 必须产生描述了所请求的改变的 `PropertyChangeEvent`, 并把那个事件传递给每个对 bean 或专门限定属性注册的 `VetoableChangeListener` 对象的 `vetoableChange()` method。如果任一个监听程序通过抛出 `PropertyVetoException` 来否决改变, bean 就必须发出另一个 `PropertyChangeEvent` 将属性恢复至原先的值, 然后它自己应该再抛出 `PropertyVetoException`。另一方面, 如果属性的改变未被否决, bean 就应该更新它的内部状态以反映此变化。如果此限定属性也是约束属性, bean 就应该在这时通知 `PropertyChangeListener` 对象。

### 支持

`java.beans.VetoableChangeSupport` 对于实现限定属性很有帮助。

## 事件

当约束和限定属性被改变时, 除了会产生 `PropertyChangeEvent` 事件之外, bean 还可以产生其他类型的事件。名为 *E* 的事件应该遵循下面这些惯例:

### 事件类

事件类应该直接或间接扩展 `java.util.EventObject`, 而且应该被命名为 `EEvent`。

### 监听程序接口

事件必须与扩展了 `java.util.EventListener` 的事件监听程序接口相关联,并且被命名为 `EventListener`。

### 监听程序 *method*

事件监听程序接口可以定义任意数量的 `method`, 这些 `method` 会接收属于 `EventListener` 类型的单一自变量并返回 `void`。

### 监听程序的注册

`bean` 必须定义一对 `method` 以供想在 `E` 事件发生时被通知的事件监听程序注册。那些 `method` 应具有以下签名:

```
public void addEventListener(EventListener)
public void removeEventListener(EventListener)
```

### 单点传播事件

单点传播事件一次只允许一个监听程序对象被注册。如果 `E` 是单点传播事件, 那么监听程序的注册 `method` 就应该具有这个签名:

```
public void addEventListener(EventListener) throws TooManyListenersException
```





# Java 开发工具

Sun 的 Java 实现包含了一些针对 Java 开发者的工具。当然，其中主要的是 Java 解释器和 Java 编译器，也还有一些其他的工具。本章说明与 JDK 一起的大部分工具。值得注意的是，本章省略了专门针对企业程序设计的 RMI 和 IDL 工具，这两个工具在《Java Enterprise in a Nutshell》(O'Reilly) 中有说明。

此处所说明的工具是 Sun 开发工具的一部分，它们是实现的细节，而不是 Java 规范本身的一部分。如果你使用 Sun JDK 之外的 Java 开发环境，就应该查阅厂商提供的工具说明文件。

本章中的一些范例对文件和路径的分隔字符使用了 Unix 惯例。如果你的开发平台是 Windows，那么就把文件名中的斜线改为反斜线，并把路径指定语句中的冒号改为分号。

apt

注释处理工具 (Annotation Processing Tool)

### 提纲 (synopsis)

```
apt [options] sourcefiles
```

### 说明

*apt* 会读取并解析指定的 *sourcefiles*。它所发现的所有注释都会被传递给适当的注释处理对象 (processor factory object)，此对象使用注释来产生以注释内容为基础的辅助源代码或数据文件，然后 *apt* 会编译 *sourcefiles* 并产生文件。

*annotation processor* 类和 *factory* 类是用 `com.sun.mirror.apt` API 以及其他 `com.sun.mirror` 的子包来定义的。



## 选项

*apt* 与 *javac* 共享了几个选项。如果命令行自变量是以 @ 开始, *apt* 就会把它当成文件, 并从那个被指定的文件中读取选项和源代码文件。相关信息请参阅 *javac*。

-Aname=value

将 name=value 对当作自变量传递给注释处理器。

-cp path

-classpath path

设定 classpath。请参阅 *javac*。

-d dir

用来放置类文件的目录。请参阅 *javac*。

-factory classname

明确地指定所要使用的注释处理 factory 的类名。

-factorypath path

指定注释处理 factory 所要搜索的路径, 而不搜索 classpath。

-help

输出辅助说明并退出。

-nocompile

告诉 *apt* 不要编译 *sourcefiles* 或任何已产生的文件。

-print

只解析指定的 *sourcefiles* 并输出它们所定义的类型提纲。不处理注释或编译任何的文件。

-s dir

指定用来存放产生的源文件的根目录。

-source version

指定所接受的语言版本。请参阅 *javac*。

-version

输出 *apt* 版本信息。

-X

显示有关非标准选项的信息。

**另可参阅**     *javac* (第四章)



---

**extcheck****JAR 版本冲突检查公用程序****提纲**

```
extcheck [-verbose] jarfile
```

**说明**

*extcheck* 会检查被包含于指定 *jarfile* 中的扩展包（或那个扩展包的较新版本）是否已安装在系统上。它通过从被指定的 *jarfile* 和在系统扩展包目录中找到的所有 JAR 文件读取 *Specification-Title* 和 *Specification-Version* 清单属性（manifest attribute）来实现。

*extcheck* 是被设计用于自动化安装命令脚本中。如果没有加上 *-verbose* 选项，它就不会输出检查的结果，而是会在指定的扩展包没有与任何已安装的扩展包冲突而且可以安全地被安装时，就将退出代码（exit code）设定为 0。如果具有相同名称的扩展包已被安装而且版本号等于或大于指定文件的版本号，它就会把退出代码设定为非零值。

**选项**

*-verbose*

输出检查到的已安装扩展包并显示检查的结果。

另可参阅 *jar*

---

**jarsigner****JAR 签名与验证工具****提纲**

```
jarsigner [options] jarfile signer  
jarsigner -verify jarfile
```

**说明**

*jarsigner* 会将数字签名加到指定的 *jarfile*，而且如果指定了 *-verify* 选项，它就会验证已添加至 JAR 文件的数字签名。所指定的 *signer* 是不区分大小写的昵名或别名，那是签名被拿来使用的项目的。所指定的 *signer* 名称会被用来查询能产生签名的私用密钥。

当你将你的数字签名用在 JAR 文件上时，实际上就是为文件的内容作了担保，你承诺 JAR 文件中只包含无恶意的程序代码、没有违反著作权的文件等。当你验证以数字方式签名的 JAR 文件时，可以判定谁是文件的签名者，并（如果验证成功）判定 JAR 文件的

内容自从加上签名后是否有更改、毁损或篡改。验证数字签名与决定是否要相信你所验证的签名所属的人或组织是完全不同的。

*jarsigner* 以及相关的 *keytool* 程序取代了 Java 1.1 中的 *javakey* 程序。

## 选项

*jarsigner* 定义了一些选项，其中有许多选项指定了找到特定签名者的密钥的方法。当使用 *-verify* 选项来验证有签名的 JAR 文件时，其实这些选项大部分都是不必要的：

*-certs*

如果此选项有与 *-verify* 或 *-verbose* 选项一起被指定，就会使 *jarsigner* 显示和签名的 JAR 文件有关的公钥证书的细节。

*-Jjavaoption*

将指定的 *javaoption* 直接传递给 Java 解释器。

*-keypass password*

指定用来加密特定 *signer* 的密钥的密码。如果没有指定此选项，*jarsigner* 就会提示你要输入密码。

*-keystore url*

*keystore* 是个包含了密钥和证书的文件。此选项指定了 *keystore* 的文件名或 URL，并被指定的 *signer* 的私用和公用密钥都会在那里被查询。默认值是用户根目录（系统属性 *user.home* 的值）中名称为 *.keystore* 的文件。这也是由 *keytool* 所管理的 *keystore* 的默认位置。

*-sigfile basename*

指定了 *.SF* 和 *.DSA* 文件的基本名称（base name），它们会被添加到 JAR 文件的 *META-INF/* 目录。如果没有指定这个选项，基本文件名就会依 *signer* 名称来决定。

*-signedjar outputfile*

指定由 *jarsigner* 所创建的已签名 JAR 文件的名称。如果未指定此选项，*jarsigner* 就会改写命令行上指定的 *jarfile*。

*-storepass password*

指定用来验证 *keystore* 完整性的密码（但不对私有密钥做加密）。如果省略此选项，*jarsigner* 就会提示你输入密码。

*-storetype type*

指定由 *-keystore* 选项所指定的 *keystore* 的类型。默认值为系统默认的 *keystore* 类型，在大部分的系统上是 Java Keystore 类型，也就是所谓的 JKS。如果你已安装了 Java Cryptography Extension，或许会想要用 JCEKS *keystore* 来表示。



-verbose

显示关于签名或验证过程的额外信息。

-verify

指定 *jarsigner* 应该验证加上签名的 JAR 文件，而不是对它签名。

另可参阅 *jar*、*keytool*、*javakey*

## jar

Java 归档工具

### 提纲

```
jar c[t|u|x[f][m][M][O][v] [jar-file] [manifest] [-C directory] [input-files]
jar i [jar-file]
```

### 说明

*jar* 是个工具，它可以创建并操作 JavaArchive (JAR) 文件。JAR 文件是个含有 Java 类文件、那些类所需要的辅助资源文件以及选择性的超元信息 (meta-information) 的 ZIP 文件。此 meta-information 包含列出 JAR 归档文件内容的清单文件，并提供关于每个文件的辅助信息。

*jar* 命令可以创建 JAR 文件、列出 JAR 文件的内容以及提取 JAR 归档文件中的文件。在 Java 1.2 及之后的版本中，它也可以增加文件到已有的归档文件，或更新归档文件的清单文件。在 Java 1.3 及之后的版本中，*jar* 也可以给 JAR 文件增加索引项。

*jar* 命令的语法会让人联想到 Unix *tar* (tape archive) 命令。*jar* 的大部分选项都是被指定为一块串接的字母，它们被当成单一自变量而不是单独的命令行自变量来传递。第一个自变量的第一个字母指定了 *jar* 所要执行的动作，这项是必要的，其他的字母是选择性的。各种文件自变量是取决于被指定的那些字母。

就和在 *javac* 中一样，任何以 @ 开头的命令行自变量都会被当成是文件的名称，其中包含一些操作或文件名。

### 命令选项

*jar* 的第一个选项的第一个字母指定了 *jar* 所要执行的基本操作。可用的选项有：

c

创建新的 JAR 归档文件。有个输入文件和 / 或目录列表必须要被指定为 *jar* 的最后一个自变量。新创建的 JAR 文件会将 *META-INF/MANIFEST.MF* 文件作为它的第一项。此自动创建的清单会列出 JAR 文件的内容，并包含每个文件的消息摘要。

i

将此 JAR 文件的内容和它在 Class-Path 清单属性中所引用的所有 JAR 文件的内容加上索引。最后产生的索引会被存储为 JAR 文件中的 *META-INF/INDEX.LIST*，而且可以被 Java 解释器或 applet viewer 用来优化其类和资源的查找算法，并避免下载不必要的 JAR 文件。此 i 选项后面必须跟着要被索引的 JAR 文件名称，其他的选项都不被允许。Java 1.3 及之后的版本提供。

t

列出 JAR 归档文件的内容。

u

更新 JAR 归档文件的内容。所有列在命令行上的文件都会被加到归档文件中。当与 m 选项一起使用时，这会向 JAR 文件增加指定的清单信息。Java 1.2 及之后的版本提供。

x

提取 JAR 归档文件的内容。命令行上所指定的文件和目录会被提取并被创建在当前的工作目录中。如果没有指定文件或目录名称，那么所有 JAR 文件中的文件和目录都会被提取。

### 修饰符选项

这四个命令中的每一个指定了字母后面可以接着额外的字母，提供关于所要执行的操作的进一步细节：

f

指出 *jar* 要操作名称被指定在命令行上的 JAR 文件。如果此选项不存在，*jar* 就会从标准输入中读取 JAR 文件，并 / 或将 JAR 文件写入标准输出。如果 f 选项存在，命令行就必须包含所要操作的 JAR 文件的名称。

m

当 *jar* 创建或更新 JAR 文件时，它会自动创建（或更新）JAR 归档文件中名为 *META-INF/MANIFEST.MF* 的清单文件。此默认清单只是列出 JAR 文件的内容。有许多的 JAR 文件会要求清单中要指定额外的信息，m 选项告诉 *jar* 命令在命令行中行指定了清单模板。*jar* 会读取此清单文件，并把它所包含的所有信息存储在它所创建的 *META-INF/MANIFEST.MF* 文件中。

M

配合 c 和 u 命令一起使用，用来告诉 *jar* 不要创建默认的清单文件。

v

告诉 *jar* 产生详细的输出。

0

配合 *c* 和 *u* 命令一起使用，用来告诉 *jar* 把文件存储在 JAR 归档文件中而不加以压缩。请注意，此选项是数字 0，而不是字母 O。

## 文件

*jar* 的第一个选项是由初始的命令字母和各种选项字母所组成。第一个选项后面接着一个文件列表：

*jar-file*

如果第一个选项包含字母 *f*，那个选项就必须接着所要创建或操作的 JAR 文件的名称。

*manifest-file*

如果第一个选项包含字母 *m*，那个选项就必须接着文件名称，其中含有清单信息。如果第一个选项同时包含字母 *f* 和 *m*，则 JAR 和清单文件就应该以与 *f* 和 *m* 选项相同的出现顺序列出。*jar* 会自动创建 JAR 文件的清单，除非指定了 *M* 选项。当 *manifest-file* 与 *m* 选项一起被指定时，除了自动产生的项之外，还应该包含要被放在清单中的额外清单项。

*files*

要被插入 JAR 归档文件或从 JAR 归档文件中提取出来的一个或多个文件和/或目录的列表。

## 额外选项

除了前面列出的所有选项之外，*jar* 也支持以下选项：

*-C dir*

用在要被处理的文件列表中，它告诉 *jar* 在处理后续的文件和目录时要更改至指定的 *dir*。后续的文件和目录名称都是相对于 *dir* 来解释的，并且会在不加上 *dir* 作为前缀的情况下被插入 JAR 归档文件中。*-C* 选项的数量不受限制，各个 *-C* 选项的能效会持续到遇到下一个 *-C* 选项。由 *-C* 选项指定的目录会以相对于当前工作目录的方式来解释，而不是由前一个 *-C* 选项所指定的目录来解释。Java 1.2 及之后的版本提供。

*-Jjavaopt*

将选项 *javaopt* 传递至 Java 解释器。



## 范例

`jar` 命令具有一组容易令人混乱的选项，但在大部分情况下，它的用法相当简单。如果要创建一个简单的 JAR 文件，它含有当前目录中所有的类文件和名称为 `images` 的子目录中的所有文件，你可以输入：

```
% jar cf my.jar *.class images
```

如果要详细地列出 JAR 归档文件的内容：

```
% jar tvf your.jar
```

如果要提取 JAR 文件的清单文件以供查看或编辑：

```
% jar xf the.jar META-INF/MANIFEST.MF
```

如果要更新 JAR 文件的清单：

```
% jar ufm my.jar manifest.template
```

另可参阅 `jarsigner`

---

## java

Java 解释器

### 提纲

```
java [ interpreter-options ] classname [ program-arguments ]  
java [ interpreter-options ] -jar jarfile [ program-arguments ]
```

### 说明

`java` 是 Java 字节码解释器，它会运行 Java 程序。要被运行的程序是由 `classname` 所指定的类。这必须是完全限定名称：它必须包含类的包名，但不能有 `.class` 扩展名。例如：

```
% java david.games.Checkers  
% java Test
```

被指定的类必须定义与以下签名完全相同的 `main()` method：

```
public static void main(String[] args)
```

此 `method` 是用来作为程序进入点：解释器会在此处开始执行。

在 Java 1.2 及之后的版本中，程序可以被封装在可执行的 JAR 文件中。要运行以这种方法封装的程序，可以使用 `-jar` 选项来指定 JAR 文件。可执行的 JAR 文件的清单必须包含 `Main-Class` 属性，它指定了 Jar 文件中哪个类包含了 `main()` method，解释器会在那里开始执行。

所有出现在要被执行的类或 JAR 文件前面的命令行选项都是针对 Java 解释器本身的选项，所有接在类名或 JAR 文件名后面的选项都是针对程序的选项。它们会被 Java 解释器忽略并被当成字符串数组传递给此程序的 `main()` method。

Java 解释器会运行到 `main()` method 后结束，所有由此程序所创建的线程（除了被标识为后台线程（daemon thread）的那些线程）也都会结束。

## 解释器版本

`java` 程序是基本版本的 Java 解释器。但是，除了这个程序之外，还有几种其他版本的 Java 解释器。这些版本都与 Java 类似，但具有专门的功能。此列表包含了所有的解释器版本，其中包括那些不再被使用的。

### `java`

这是基本版本的 Java 解释器。通常使用这个就行了。

### `javaw`

这个版本的解释器只有 Windows 平台有。当你想运行 Java 程序（例如脚本）但又不不要创建控制台窗口时，就使用 `javaw`。

### 客户端或服务端 VM

Sun 的“热点 (HotSpot)”虚拟机有两种版本：一种是供暂时存在的客户端应用程序使用，而另一种是供长时间运行的服务器程序代码使用。在 Java 1.4 中，你可以用 `-server` 选项来选择服务器版本的 VM，可以用 `-client` 选项来指定客户端 VM（这是默认值）。在 Java 5.0 中，如果解释器检测到它是运行于“服务器等级”的硬件上（通常是具有多个 CPU 的计算机），就会自动进入服务器模式。

## 原有的解释器版本

### `oldjava`

这个版本的解释器被包含在 Java 1.2 和 Java 1.3 中，这是为了保证对 Java 1.1 解释器的兼容性。它用 Java 1.1 的类载入模式（class-loading scheme）来载人类。很少有 Java 应用程序会必须使用这个版本的解释器，而它在 Java 1.4 中已被撤除。

### `oldjavaw`

在 Java 1.2 和 1.3 中，这个版本的解释器只有 Windows 平台才有，它结合了 `oldjava` 和 `javaw` 的特性。

### `java_g`

在 Java 1.0 和 Java 1.1 中，`java_g` 是调试版本的 Java 解释器，它包含了一些专门的命令行选项。Windows 平台也有个 `javaw_g` 程序。`java_g` 没有被包含在 Java 1.2 及之后的版本中。

### 标准 VM

在 Java 1.3 中, 你可以用 `-classic` 选项来指定你要使用“标准 VM”(在本质上与 Java 1.2 VM 相同) 来代替 HotSpot VM (它使用了渐增式编译 (incremental compilation))。这个选项在 Java 1.4 中已被移除。

### 实时编译器 (Just-in-time compiler)

在 Java 1.2 和 Java 1.3 中, 当你指定 `-classic` 选项时, Java 解释器会使用实时编译器 (如果你的平台提供)。JIT 会在运行时把 Java 字节码转换为原生机器指令, 并大幅加速典型 Java 程序的运行。如果你不想使用 JIT, 可以通过将 `JAVA_COMPILER` 环境变量设定为“NONE”, 或使用 `-D` 选项将 `java.compiler` 系统属性设定为“NONE”来停用 JIT。

```
% setenv JAVA_COMPILER NONE // Unix csh 语法
% java -Djava.compiler=NONE MyProgram
```

如果你想使用不同的 JIT 编译器实现, 可以把环境变量或系统属性设定为所要实现的名称。此环境变量和属性在 Java 1.4 中已不再被使用, Java 1.4 使用了含有高效率的 JIT 技术的 HotSpot VM。

### 线程系统

在 Solaris 和相关的 Unix 平台上, 你可以选择由 Java 1.2 解释器和 Java 1.3 的“标准 VM”所使用的线程类型。如果要使用原生 OS 线程, 可以指定 `-native`; 如果要使用非原生线程 (或 green thread, 此为默认值), 可以指定 `-green`。在 Java 1.3 中, 默认的“客户端 VM”使用了原生线程。在 Java 1.3 中指定了 `-green` 或 `-native` 也就暗含指定了 `-classic`。这些选项在 Java 1.4 中已不再 (或不需要) 被支持。

### 常用选项

以下是最常被使用的选项。

#### `-classpath path`

指定在尝试加载类时 Java 所要搜索的目录和 JAR 文件。在 Java 1.2 及之后的版本中, 此选项只会指定应用程序类的位置。在 Java 1.0 和 1.1 中, 如果使用 `oldjava` 解释器, 此选项会指定系统类、扩展类以及应用程序类的位置。

#### `-cp`

`-classpath` 的同义词。Java 1.2 及之后的版本提供。

#### `-Dpropertyname=value`

将系统属性列表中的 `propertyname` 定义为等于 `value`。你的 Java 程序可以通过属性名称来查询被指定的值。你可以指定任意数量的 `-D` 选项。例如:



- `% java -Dawt.button.color=gray -Dmy.class.pointsize=14 my.class`
- `-fullversion`  
列出完整的 Java 版本字符串并退出，其中包括编译编号 (build number)。可与 `-version` 作比较。
- `-help, -?`  
输出使用信息并退出。请参阅 `-X`。
- `-jar jarfile`  
运行指定的可执行文件 *jarfile*。被指定的 *jarfile* 的清单必须包含一个 `Main-Class` 属性，以识别出具有 `main() method` 的类，程序会在那里开始运行。Java 1.2 以及之后的版本提供。
- `-showversion`  
功能类似 `-version` 选项，除了解释器会在输出版本信息之后继续运行。Java 1.3 及之后的版本提供。
- `-version`  
输出 Java 解释器的版本并退出。
- `-X`  
显示非标准解释器选项（那些会以 `-X` 开始）的使用信息并退出。Java 1.2 及之后的版本提供。
- `-Xbootclasspath:path`  
指定由目录、ZIP 文件和 JAR 文件所组成的搜索路径，*java* 解释器应该要用它来查询系统类。此选项的使用非常罕见。Java 1.2 及之后的版本提供。
- `-Xbootclasspath/a:path`  
将指定的 *path* 添加到系统 *classpath*。Java 1.3 及之后的版本提供。
- `-Xbootclasspath/p:path`  
将指定的 *path* 默认为系统启动 *classpath*。Java 1.3 及之后的版本提供。

## Assertion 选项

以下选项指定 *assertion* 是否被测试以及在何处被测试。这些选项是在 Java 1.4 中加入的。

`-disableassertions[:where]`

停用 *assertion*。它是 Java 1.4 中新出现的，而且可以缩写为 `-da`。在单独使用时，它会停用所有的 *assertion*（除了在系统类中的那些），这是默认值。如果要停用单一类中的 *assertion*，可以在选项后面加上冒号及完全限定类名。如果要停用整个包

(以及它所有的子包)中的 `assertion`, 可以在此选项后面加上冒号、包的名称以及三个点号。请参阅 `-enableassertions` 和 `-disablesystemassertions`。

`-da[:where]`

停用 `assertion`。请参阅 `-disableassertions`。

`-disablesystemassertions`

停用所有系统类中的 `assertion` (这是默认值)。它可以被缩写为 `-dsa` 而且没有任何选项。

`-dsa`

`-disablesystemassertions` 的缩写。

`-enableassertions[:where]`

启用 `assertion`。此选项可被缩写为 `-ea`。在单独使用时, 它会启用所有的 `assertion` (除了在系统类中的)。如果要启用单一类中的 `assertion`, 可以在选项后面加上冒号和完整类名。如果要启用整个包 (及其子包) 中的 `assertion`, 可以在选项后面加上冒号、包名以及三个点号。请参阅 `-disableassertions` 和 `-enablesystemassertions`。

`-ea[:where]`

启用 `assertion`。这是 `-enableassertions` 的缩写。

`-enablesystemassertions`

启用所有系统类中的 `assertion`。可被缩写为 `-esa`。

`-esa`

`-enablesystemassertions` 的缩写。

## 性能调整选项

以下选项会选择要运行哪个版本的 VM, 并微调其内存配置、内存回收以及渐增式编译。以 `-X` 开始的选项是非标准的, 可能会因版本不同而异。

`-classic`

运行“标准 VM”来代替默认的高性能“客户端 VM”。只有 Java 1.3 才提供。

`-client`

针对典型的客户端应用程序来优化 HotSpot VM 的渐增式编译。此选项通常会延迟一些编译, 以支持较快速的应用程序启动时间。Java 1.4 及之后的版本提供。请参阅 `-server` 选项。

**-d32**

以 32 位模式运行。此选项在 Java 1.4 及之后的版本中是有效的，但现在只有针对 Solaris 平台实现。

**-d64**

以 64 位模式运行。此选项在 Java 1.4 及之后的版本中是有效的，但现在只有针对 Solaris 平台实现。

**-green**

在像 Solaris 和 Linux 这种支持多种线程风格的操作系统上选择非原生（或 green）线程。这是 Java 1.2 的默认值。在 Java 1.3 中，可使用这个选项也可使用 -classic 选项。请参阅 -native。只有 Java 1.2 和 1.3 提供。

**-native**

在像 Solaris 和 Linux 这种支持多种线程风格的操作系统上选择原生线程，而不是默认的 green 线程。在某些环境中（例如在多 CPU 的计算机上运行时）使用原生线程很有好处。在 Java 1.3 中，默认的 HotSpot 虚拟机使用了原生线程。只有 Java 1.2 和 1.3 提供。

**-server**

针对服务器等级的应用程序来优化 VM 的渐增式编译。通常，此选项会造成较长的启动时间，但会有较好的后续性能。Java 1.4 及之后的版本提供。在 Java 5.0 及之后的版本中，如果是在“服务器等级”的硬件上运行（例如双 CPU 的机器），则有许多 VM 会自动选择此选项。请参阅 -client。

**-Xbatch**

告诉 HotSpot VM 要在前台执行所有的实时编译，而不考虑编译所需的时间。如果没有加上这个选项，VM 就会在前台作解释时在后台编译这些 method。Java 1.3 及之后的版本提供。

**-Xincgc**

使用渐增式的内存回收。在此模式中，内存回收程序会在后台连续运行，而运行中的程序很少会在内存回收发生时出现明显的停顿。但是，使用此选项通常会造成整体性能降低 10%。Java 1.3 及之后的版本提供。

**-Xint**

告诉 HotSpot VM 只以解释模式运行，而不执行任何实时的编译。Java 1.3 及之后的版本提供。

**-Xmixed**

告诉 HotSpot VM 在常被使用的 method（“hotspots”）上执行实时编译，而以解释



模式执行其他的 `method`。这是默认的行为。可与 `-Xbatch` 和 `Xint` 作比较。Java 1.3 及之后的版本提供。

`-Xms initmem[k|m]`

指定当解释器启动时，要分配多少内存给堆。依默认，`initmem`会以字节为单位来指定。你可以通过添加字母 `k` 来以千字节指定，或是添加字母 `M` 来以兆字节指定。默认值是 2 MB。对于大型或对内存需要较多的应用程序（例如 Java 编译器），你可以通过启动具有较大数量内存的解释器来改善运行时性能。初始堆的大小至少必须指定为 1 MB。Java 1.2 及之后的版本提供。在 Java 1.2 之前，要使用 `-ms`。

`-Xmxmaxmem[k|m]`

指定针对动态分配的对象和数组，解释器所能使用的最大堆的大小。`maxmem`根据默认是以字节为单位来指定。你可以通过添加字母 `k` 来以千字节指定 `maxmem`，或是通过添加字母 `M` 来以兆字节指定 `maxmem`。默认值为 64 MB。堆的大小无法被指定为小于 2 MB。Java 1.2 及之后的版本提供。在 Java 1.2 之前，要使用 `-mx`。

`-Xnoclassgc`

不要对类进行内存回收。Java 1.2 及之后的版本提供。在 Java 1.1 中要使用 `-noclassgc`。

`-Xsssize[k|m]`

以字节、千字节或兆字节来设定线程堆的大小。Java 1.3 及之后的版本提供。

## Instrumentation 选项

以下选项支持调试、剖析 (profiling) 以及其他的 VM。以 `x` 开头的选项是非标准的，而且可能会随版本而异。

`-agentlib:agent[=options]`

这是 Java 5.0 中新出现的，此选项指定了要与解释器一起启动的 JVMTI 代理程序以及针对代理程序的选项。JVMTI 就是 Java 虚拟机工具接口 (Java Virtual Machine Tool Interface)，并为未来版本中的 JVMDI 和 JVMPI (调试与剖析接口) 打基础。这代表一般的 `-agentlib` 选项会取代工具所特有的选项（例如 `-Xdebug` 和 `-Xrunhprof`）。范例：

```
% java -agentlib:hprof=help
% java -agentlib:jdwp=help
```

`-agentpath:path-to-agent[=options]`

就和 `-agentlib` 一样，但具有针对代理程序函数库的明确指定路径。Java 5.0 及之后的版本提供。

#### -debug

使 *java* 能以允许 *jdb* 调试程序将自己添加到解释器部分的方式来启动。在 Java 1.2 及之后的版本中, 这个选项已被 *-xdebug* 取代。

#### -javaagent:jarfile[=options]

当解释器启动时加载 Java 语言 instrumentation 代理程序。被指定的 *jarfile* 必须要有包含了 Agent-Class 属性的清单。此属性必须命名一个类, 其中包含了代理程序的 *premain()* method。所有的选项都会与 *java.lang.instrument.Instrumentation* 对象一起被传递给这个 *premain()* method。进一步的细节请参阅 *java.lang.instrument*。

#### -verbose, -verbose:class

每当 *java* 加载类时就输出消息。在 Java 1.2 及之后的版本中, 你可以用 *-verbose:class* 作为同义词。

#### -verbose:gc

当内存回收发生时就输出消息。Java 1.2 及之后的版本提供。在 Java 1.2 之前, 可以使用 *-verbosegc*。

#### -verbose:jni

当原生 method 被调用时就输出消息。Java 1.2 及之后的版本提供。

#### -Xcheck:jni

当使用 Java Native Interface 功能时就执行额外的验证检查。Java 1.2 及之后的版本提供。

#### -Xdebug

以允许调试程序与解释器通信的方式来启动解释器。Java 1.2 及之后的版本提供。在 Java 1.2 之前, 可以使用 *-debug*。这在 Java 5.0 中已过时, 要改为使用 *-agentlib* 选项。

#### -Xfuture

严格地检查所有被载入的类文件的格式。如果没有加上这个选项, *java* 就会执行与 Java 1.1 所执行的相同检查。Java 1.2 及之后的版本提供。

#### -Xloggc:filename

以加上时间戳的方式将内存回收事件记录到具名文件。

#### -Xprof

将概要输出以标准输出形式输出。Java 1.3 及之后的版本提供。在 Java 1.2 中或当使用 *-classic* 选项时, 就使用 *-Xrunhprof*。在 Java 1.2 之前, 要使用 *-prof*。

`-Xrunhprof:suboptions`

开启 CPU、堆或监控程序。*suboptions* 是以逗号分隔、由 *name=value* 对组成的列表。可用 `-Xrunhprof:help` 来得知被支持的选项和值的列表。Java 1.2 及之后的版本提供。这在 Java 5.0 中已过时，要改用 `-agentlib` 选项。

## 高级选项

Java 解释器也支持少数以 `-xx` 开始的高级配置选项。这些选项依赖于版本和平台，而 Sun 的说明文件把它们描述成“不建议随便使用”。但是，如果你想微调实际环境的线程、内存配置、内存回收、信号处理或实时编译性能，或许就会对它们有兴趣。请访问 <http://java.sun.com/docs/hotspot/>。

## 载入类

Java 解释器知道要到哪里找出组成 Java 平台的系统类。在 Java 1.2 及之后的版本中，它也知道要到哪里找出安装在系统扩展包目录中的所有扩展包的类。但是，解释器必须被告知要到何处找出组成所要运行的应用系统的非系统类。

类文件被存储在对应它们的包名的目录中。例如，类 `com.davidflanigan.utils.Util` 是被存储在 `com/davidflanigan/utils/Util.class` 文件中。默认情况下，解释器会使用当前工作目录作为根目录并搜索在此目录中及目录下的所有类。

解释器也可以搜索 ZIP 和 JAR 文件中的类。如果要告诉解释器在哪里搜索类，可以指定 *classpath*：这是由目录、ZIP 和 JAR 归档文件所组成的列表。在寻找类时，解释器依各位置的指定顺序来逐一搜索。

指定 *classpath* 的最简单方法就是设定 `CLASSPATH` 环境变量，这非常像由 Unix shell 或 Windows 命令解释路径所使用的 `PATH` 变量。如果在 Unix 中要指定 *classpath*，可以输入像这样的命令：

```
% setenv CLASSPATH ~/.myclasses:/usr/lib/javautils.jar:/usr/lib/javaapps
```

在 Windows 系统上，可以使用如下的命令：

```
C:\> set CLASSPATH=.;c:\myclasses;c:\javatools\classes.zip;d:\javaapps
```

请注意，Unix 和 Windows 使用了不同的字符来分隔目录和路径元素。

你也可以用 `-classpath` 或 `-cp` 选项来对 Java 解释器指定 *classpath*。以这些方式所指定的路径会覆盖由 `CLASSPATH` 环境变量所指定的路径。在 Java 1.2 及之后的版本中或



在使用 *oldjava* 解释器时，这个选项指定了针对所有类的搜索路径，其中包括了系统类和扩展类。

另可参阅 *javac*、*jdb*

## javac

Java 编译器

### 提纲

```
javac [ options ] files
```

### 说明

*javac* 是 Java 编译器，它会将 Java 源代码（在 *.java* 文件中）编译为 Java 字节码（在 *.class* 文件中）。Java 编译器本身是用 Java 编写的。

*javac* 可以被传递任意数量的 Java 源文件（source file），这些文件的名称都必须以 *.java* 扩展名结束。*javac* 会为每个定义于源文件的类产生单独的 *.class* 类文件。每个源文件可以包含任意数量的类，虽然只有一个可以是公共的顶层类。源文件的名称（不含 *.java* 扩展名）必须与它所包含的公共类的名称相匹配。

在 Java 1.2 及之后的版本中，如果在命令行上指定的文件名是以字符 @ 开始，那个文件就不会被当成 Java 源文件，而是会被当成由编译器选项和 Java 源文件组成的列表。因此，如果你把针对特殊项目的 Java 源文件列表保留在名为 *project.list* 的文件中，就可以用以下命令一次编译所有文件：

```
% javac @project.list
```

如果要编译源文件，*javac* 就必须能找到所有用于源文件中的类的定义。它会在源文件和类文件这两种形式中寻找定义，自动编译所有不具有对应类文件或在最近编译过后又被修改的源文件。

### 常用选项

最常被使用的编译选项如下：

*-classpath path*

指定路径，供 *javac* 用来查询特定源代码中所引用的类。这个选项会覆盖所有由 CLASSPATH 环境变量所指定的路径。所指定的 *path* 是由目录、ZIP 文件以及 JAR 归档文件所组成的有序列表，在 Unix 系统上会以冒号分隔，在 Windows 系统上会

以分号分隔。如果 `-sourcepath` 选项没有被设定, 那么这个选项也会指定针对源文件的搜索路径。

#### `-d directory`

指定用来存储类文件的目录 (或在哪个目录下)。默认情况下, `javac` 会把它产生的 `.class` 文件存储在与定义有那些类的 `.java` 文件相同的目录中。但是, 如果 `-d` 选项被指定了, 所指定的 `directory` 就会被当成类层次的根, 而 `.class` 文件会被放在这个目录中或在其下的适当子目录中, 这取决于类的包名。因此, 有以下命令:

```
% javac -d /java/classes Checkers.java
```

如果 `Checkers.java` 文件没有 `package` 语句, 文件 `Checkers.class` 就会被放在 `/java/classes` 目录中。另一方面, 如果源文件是在包中:

```
package com.davidflanagan.games;
```

`.class` 文件就会被存储在 `/java/classes/com/davidflanagan/games` 中。当 `-d` 选项被指定时, `javac` 就会自动创建它所需要的目录来把它的类文件存储在适当的地方。

#### `-encoding encoding-name`

当字符编码名称与默认的平台编码不同时, 可用此选项来指定源文件所使用的字符编码名称。

#### `-g`

告诉 `javac` 要对输出类文件加上行号、源文件以及局部变量信息, 以供调试程序使用。默认情况下, `javac` 只会产生行号。

#### `-g:none`

告诉 `javac` 把调试信息加在输出类文件中。Java 1.2 及之后的版本提供。

#### `-g:keyword-list`

告诉 `javac` 输出调试信息的类型, 这些是由以逗号分隔的 `keyword-list` 所指定的。有效的关键字是: `source`, 它指定了源文件信息; `lines`, 指定了行号信息; `vars`, 指定了局部变量调试信息。Java 1.2 及之后的版本提供。

#### `-help`

输出选项列表。请参阅 `-X`。

#### `-Jjavaoption`

将自变量 `javaoption` 直接传给 Java 解释器。例如: `-J-Xmx32m`。 `javaoption` 不能包含空格; 如果有多个自变量要传给解释器, 可以使用多个 `-J` 选项。Java 1.1 及之后的版本提供。

#### `-source release-number`

指定编写程序代码所用的 Java 版本。合法的 `release-number` 值是 5、1.5、1.4 和 1.3。选项 5 和 1.5 是同义的而且是默认值：编译器接受所有 Java 5.0 语言特性。使用 `-source 1.4` 可以让编译器忽略 Java 5.0 语言特性，例如 `enum` 关键字。使用 `-source 1.3` 可以让编译器忽略 Java 1.4 引入的 `assert` 关键字。此选项在 Java 1.4 及之后的版本中提供。

#### `-sourcepath path`

指定目录、ZIP 文件以及 JAR 归档文件列表，以供 `javac` 在寻找源文件时搜索。在这个源路径中找到的文件如果没有相对类文件被找到或是源文件比类文件新，就会被编译。默认情况下，源文件会与类文件在同一个地方被搜索到。Java 1.2 及之后的版本提供。

#### `-verbose`

告诉编译器显示关于进行中事务的信息。尤其是它会使 `javac` 列出所有它编译的源文件，其中包括没有出现在命令行上的文件。

#### `-X`

告诉 `javac` 编译器显示非标准选项（那些都会以 `-X` 开头）的使用信息。Java 1.2 及之后的版本提供。

## 警告选项

以下选项控制了 `javac` 所产生的警告信息：

#### `-deprecation`

告诉 `javac` 每当使用到过时的 API 时就产生相应的警告信息。默认情况下，`javac` 只会对每个使用了过时 API 的文件提出一条警告信息。Java 1.1 及之后的版本提供。在 Java 5.0 中，这是 `-Xlint:deprecation` 的同义词。

#### `-nowarn`

告诉 `javac` 不要输出警告信息。但错误仍要报告。

#### `-Xlint`

启用所有与“lint”程序有关的建议警告。在使用这个选项时，以下所有详细说明了警告都是被建议的。

#### `-Xlint:warnings`

启用或停用以逗号分隔的具名警告类型列表。在使用这个选项时，可用的警告类型如下。具名警告可以通过在它前面加上减号来暂停：



`all`

启用所有的 lint 警告。

`deprecation`

对使用过时的 API 提出警告。请参阅 `-deprecation`。

`fallthrough`

当 `switch` 语句中的一个 `case` “落到 (falls through)” 下一个 `case` 时提出警告。请参阅 `-Xswitchcheck`。

`finally`

当 `finally` 子句无法正常结束时提出警告。

`path`

当命令行上某处所指定的路径目录不存在时提出警告。

`serial`

如果 `Serializable` 类不具有 `serialVersionUID` 字段就提出警告。

`unchecked`

对每个未校验的泛型的使用提供详细的警告信息。

`-Xmaxerrors num`

列出的错误不要多于 `num` 个。

`-Xmaxwarns num`

列出的警告不要多于 `num` 个。

`-Xstdout filename`

告诉 `javac` 将警告和错误信息传送到指定的文件，而不是将它们写入平台。Java 1.4 及之后的版本提供。

`-Xswitchcheck`

对 `switch` 语句中 “fall through” 的 `case` 子句提出警告。在 Java 5.0 中，要使用 `-Xlint:fallthrough`。

## 交叉编译选项

以下选项在使用 `javac` 来编译倾向于在不同 Java 版本下运行的类文件时很有用：

`-bootclasspath path`

指定 `javac` 用来查找系统类的路径。此选项没有指定用来运行编译器本身的系统类，只有编译器所读取的系统类。Java 1.2 及之后的版本提供。

`-endorseddirs path`

覆盖目录以搜索已签署的标准 JAR 文件。

`-extdirs path`

指定一个目录列表以搜索标准扩展功能的 JAR 文件。Java 1.2 及之后的版本提供。

`-target version`

指定产生的类文件所要使用的类文件格式。`version` 可以是 1.1、1.2、1.3、1.4、1.5 或 5。选项 1.5 和 5 是同义的而且在 Java 5.0 中是默认值，除非指定了 `-source 1.4`，在这样的情况下，`-target 1.4` 就是默认值。使用这个标记可以设定类文件的版本编号，以让最后产生的类文件不被先前版本的 VM 执行。

`-Xbootclasspath:path`

`-bootclasspath` 的替代方案。

`-Xbootclasspath/a:path`

将指定的路径附加至 `bootclasspath`。Java 1.3 及之后的版本提供。

`-Xbootclasspath/p:path`

将指定的路径加在 `bootclasspath` 前面。

## 环境

### CLASSPATH

指定由目录、ZIP 文件以及 JAR 归档文件所组成的有序列表，`javac` 应该要在其中寻找用户类文件和源文件。此变量可以由 `-classpath` 选项覆盖。

另可参阅 `java`、`jdb`

## javadoc

Java 说明文件管理器

### 提纲

```
javadoc [ options ] @list package... sourcefiles...
```

### 说明

`javadoc` 可以为任意数量所指定的包和类产生 API 说明文件。`javadoc` 命令行可以列出任意数量的包名和任意数量的 Java 源文件。为了方便起见，在处理大量的命令行选项或大量的包和类名时，你可以把它们全部放在辅助文件中，并在命令行指定的那个文件名称的前面加上 `@` 字符。

*javadoc* 使用 *javac* 编译器来处理所有被指定的 Java 源文件和指定包中的所有 Java 源文件。它使用从这个过程中收集到的信息来产生详细的 API 说明文件。最重要的是，产生的说明文件包含了所有定义于源文件中的说明注释内容。关于在你自己的 Java 程序代码中编写注释的信息，请参阅第七章。

当你为 *javadoc* 指定要处理的 Java 源文件时，必须要指定包含了完整的文件路径的文件名称。但是，更常见的是使用 *javadoc* 来建立针对整个包的类别的说明文件。当你指定 *javadoc* 所要处理的包时，要指定包名，而不是指定包含有那个包的源代码的目录。在这样的情况下，你可能必须指定 *-sourcepath* 选项，以让源代码不是存储在已列出的默认 *classpath* 的位置时，*javadoc* 可以正确地找到你的包的源代码。

*javadoc* 默认情况下会创建 HTML 说明文件，但你可以通过定义能以你所要的格式来产生说明文件的 *doclet* 类来自定义它的行为。你可以使用由 *com.sun.javadoc* 包定义的 *doclet* API 来编写你自己的 *doclet*。此包的说明文件被包含在 Java 1.2 及之后版本中的标准说明文件中。

*javadoc* 在 Java 1.2 中得到了重要的新功能。我们在这说明此程序的 Java 1.2 及之后的版本，但不区分各版本之间的特性差异。

## 选项

*javadoc* 定义了大量的选项。有些是 *javadoc* 一定能识别的标准选项，其他选项则是由产生说明文件的 *doclet* 所定义。针对标准 HTML *doclet* 的选项包含于以下列表中：

### -1.1

模拟 Java 1.1 版的 *javadoc* 的输出风格和目录结构。此选项存在于 Java 1.2 和 1.3，但在 Java 1.4 中已移除。

### -author

在产生的说明文件中加上由 *@author* 所指定的作者信息。只有默认的 *doclet* 提供。

### -bootclasspath

指定系统类的位置。这在交叉编译时很有用。关于此选项的更多信息，请参阅 *javac*。

### -bottom text

在每个产生的 HTML 文件底端显示 *text*，可以包含 HTML 标记。请参阅 *-footer*。只有默认的 *doclet* 提供。



**-breakiterator**

使用 `java.text.BreakIterator` 算法来决定 doc 注释中总结句子的结尾。只有默认的 doclet 提供。

**-charset encoding**

指定针对输出的字符编码。当然，这取决于你的源代码的说明注释中所使用的编码。`encoding` 值会被用在 HTML 输出中的 `<meta>` 标记中。只有默认的 doclet 提供。

**-classpath path**

指定一个路径让 `javadoc` 查询类文件和源文件（如果你没有指定 `-sourcepath` 选项）。因为 `javadoc` 使用 `javac` 编译器，所以它必须要能找到要被说明的包所引用的所有类的类文件。关于此选项以及由 `CLASSPATH` 环境变量所提供的默认值的更多信息，请参阅 `java` 和 `javac`。

**-d directory**

指定 `javadoc` 用来存储所产生的 HTML 文件的目录。如果忽略此选项，就会使用当前目录。只有默认的 doclet 提供。

**-docencoding encoding**

指定要被用于输出 HTML 文档的编码。此处所指定的编码名称可能不会与以 `-charset` 选项指定的字符集名称相匹配。只有默认的 doclet 有提供。

**-docfilessubdirs**

以递归方式复制 `doc-files` 目录的所有子目录，而不是只直接复制包含于 `doc-files` 中的文件。只有默认的 doclet 提供。

**-doclet classname**

指定用来产生说明文件的 doclet 类的名称。如果没有指定此选项，`javadoc` 就会使用默认的 HTML doclet 来产生说明文件。

**-docletpath classpath**

如果由 `-doclet` 标记所指定的类无法从默认的 `classpath` 获得时，系统可以从这个选项指定的路径进行加载。

**-doctitle text**

提供一个标题来显示在说明文件概述文件的开头。此文件通常是读者在浏览产生的说明文件时第一个看到的东西。标题可以包含 HTML 标记。只有默认的 doclet 提供。

**-encoding encoding-name**

指定输入源文件和它们所包含的说明注释的字符编码。它可以与 `-docencoding` 所指定的输出编码不同。默认值为平台的默认编码方式。

`-exclude packages`

从由 `-subpackages` 选项定义的包集合中排除特定名称的包。`packages` 是个以冒号分隔的包名列表。只有默认的 doclet 提供。

`-excludedocfilessubdir dirs`

在指定了 `-docfilessubdirs` 时，排除 `doc-files` 目录的指定子目录。这在排除版本控制目录时很有用。`dirs` 是以冒号分隔、相对于 `doc-files` 目录的目录名称列表。只有默认的 doclet 提供。

`-extdirs dirlist`

指定用来搜索标准扩展包的目录列表。只有在以 `-bootclasspath` 来进行交叉编译时才需要。相关细节请参阅 `javac`。

`-footer text`

指定要显示在每个文件底部导航条右边的文字。`text` 可以包含 HTML 标记。请参阅 `-bottom` 和 `-header`。只有默认的 doclet 提供。

`-group title packagelist`

`javadoc` 会产生顶层的概述页面，列出生成说明文件中的所有包，默认情况下这些包会依字母顺序列在一个表格中。但是，你可以用这个选项把它们拆散为由相关包组成的组。`title` 指定了包组的标题，例如“Core Packages”。`packagelist` 是以冒号分隔的由包名组成的列表，其中每个名称都可以在结尾加上 `*` 字符。`javadoc` 命令行可以包含任意数量的 `-group` 选项。例如：

```
% javadoc -group "AWT Packages" java.awt* \
-group "Swing Packages" javax.accessibility:javax.swing*
```

`-header text`

指定在接近文件顶部、上方导航条的右边所要显示的文字。`text` 可以包含 HTML 标记。请参阅 `-footer`、`-doctitle` 以及 `-windowtitle`。只有默认的 doclet 提供。

`-help`

显示 `javadoc` 的使用说明。

`-helpfile file`

指定 HTML 文件的名称，其中包含关于生成的说明文件的使用辅助说明。`javadoc` 在所有生成文件中都包含了对辅助说明文件的链接。如果没有指定这个选项，`javadoc` 就会创建默认的辅助说明文件。只有默认的 doclet 提供。

`-Jjavaoption`

将自变量 `javaoption` 直接传递给 Java 解释器。在处理大量的包时，你可能会需要使用这个选项来增加 `javadoc` 允许使用的内存量。例如：

```
% javadoc -J-Xmx64m
```

请注意，因为 `-J` 选项会在 *javadoc* 启动之前被直接传给 Java 解释器，所以它们不可以被包括在命令行上以 `@list` 语法指定的外部文件中。

`-keywords`

告诉 *javadoc* 要把类型和成员名称包含在 `<Meta>` 标记关键字列表中。只有默认的 doclet 提供。

`-link url`

指定另一个由 *javadoc* 生成的说明文件的顶层目录的绝对或相对 URL。*javadoc* 使用这个 URL 作为基准 URL，以从当前文件链接至当前文件中未说明的包、类、method 和字段。例如，在使用 *javadoc* 来产生你自己的包的说明文件时，你可以用这个选项把你的说明文件链接至核心 Java API 的 *javadoc* 说明文件。只有默认的 doclet 提供。

由 *url* 所指定的目录必须包含一个名为 *package-list* 的文件，而且 *javadoc* 必须要能在运行时读取这个文件。这个文件会由先前运行的 *javadoc* 自动产生，它包含了所有说明于 *url* 下的包的列表。

`-link` 选项可以被指定多次，虽然这在较早的 Java 1.2 版本中并不能正常运行。如果没有指定 `-link`，那么在产生的说明文件中，对外部的类和成员的引用就不会被超级链接。

`-linkoffline url packagelist`

与 `-link` 选项类似，除了 *packagelist* 文件会被明确地在命令行上指定之外。当 *url* 所指定的目录没有 *package-list* 文件或那个文件在 *javadoc* 运行时无法取用时，这个选项很有用。只有默认的 doclet 提供。

`-linksource`

创建每个源文件的 HTML 版本并在说明文件页面加上针对这些 HTML 的链接。只有默认的 doclet 提供。

`-locale language_country_variant`

指定生成说明文件的位置 (locale)。它被用来查询资源文件 (resource file)，其中包含针对输出文件的本地化信息和文字。

`-nocomment`

忽略所有的 doc 注释并产生只包含原始 API 信息但没有任何相关信息的说明文件。只有默认的 doclet 提供。



-nodeprecated

告诉 *javadoc* 忽略掉已过时特性的说明文件。此选项暗示着 *-nodeprecatedlist*。只有默认的 doclet 提供。

-nodeprecatedlist

告诉 *javadoc* 不要产生 *deprecated-list.html* 文件而且不要输出链接到导航条上。只有默认的 doclet 提供。

-nohelp

告诉 *javadoc* 不要在导航条上产生针对它的辅助说明文件或链接。只有默认的 doclet 提供。

-noindex

告诉 *javadoc* 不要产生索引文件。只有默认的 doclet 提供。

-nonavbar

告诉 *javadoc* 省略每个文件顶端和底端的导航条,也省略由 *-header* 和 *-footer* 指定的文字。这在产生要被打印的说明文件时很有用。只有默认的 doclet 提供。

-noqualifier *packages* | *all*

*javadoc* 会针对同一个要说明的包中的类,在它产生的说明文件中忽略包名。此选项告诉它要额外忽略特定包中的类的包名,如果使用了关键字 *all*,就是忽略所有的包名。*packages* 是以冒号分隔的包名列表,它可以包含 \* 通配符来指出子包。例如, *-noqualifier java.io:java.nio.\** 会排除在 *java.io* 包和 *java.nio* 及其子包中所有类的包名。只有默认的 doclet 提供。

-nosince

忽略 doc 注释中的 @since 标记。只有默认的 doclet 提供。

-notimestamp

不要输出时间戳到 HTML 注释中。只有默认的 doclet 提供。

-notree

告诉 *javadoc* 不要产生 *tree.html* 类层次结构图或在导航条中提供链接。只有默认的 doclet 提供。

-overview *filename*

从 *filename* 读取概要 doc 注释并在概要页中使用那个注释。此文件没有包含 Java 源代码,所以 doc 注释不应该真的出现在 */\*\** 和 *\*/* 定界符之间。

-package

在输出中加上包层次可见的类和成员以及 *public* 与 *protected* 类和成员。

**-private**

在产生的说明文件中加上所有的类和成员，其中包括 `private` 和包层次可见的类和成员。

**-protected**

在产生的输出中加上 `public` 和 `protected` 类与成员。这是默认值。

**-public**

在产生的输出中只加上 `public` 类和成员。省略了 `protected`、`private` 以及包层次可见的类和成员。

**-quiet**

抑制除了警告和错误消息之外的输出。

**-serialwarn**

对没有适当地以 `@serial` 和相关 doc 注释标记来说明序列化格式的可序列化类提出警告。只有默认的 doclet 提供。

**-source release**

指定编写源文件所使用的 Java 版本。请参阅 `javac` 的 `-source` 选项。合法的值有 5、1.5、1.4 和 1.3。选项 1.5 和 5 是同义的，而且是默认值。

**-sourcepath path**

指定源文件的搜索路径，通常会被设定为单一根目录。`javadoc` 会在寻找实现了指定包的 Java 源文件时使用这个路径。

**-splitindex**

产生多个索引文件，每个字母对应一个索引文件。在说明大量程序代码时会用到这个选项。否则，由 `javadoc` 产生的单一索引文件会因太庞大而难以使用。只有默认的 doclet 提供。

**-stylesheetfile file**

指定一个文件以用来为生成的 HTML 做 CSS 样式表。`javadoc` 会在产生的说明文件中插入对此文件的适当链接。只有默认的 doclet 提供。

**-subpackages packages**

指定 `javadoc` 应该处理特定的包及其全部子包。`packages` 是以冒号分隔、由包名或包名前缀所组成的列表。使用此选项通常会比明确地列出所有需要的包名来得简单。例如：

```
-subpackages java:javax
```

请参阅 `-exclude`。只有默认的 doclet 提供。

`-tag tagname:where:header-text`

指定 *javadoc* 应该通过输出文字 *header-text*、后面接着标记的文字，来处理名为 *tagname* 的 doc 注释标记。这使得在 doc 注释中使用简单的自定义标记（使用与 `@return` 和 `@author` 相同的语法）成为可能。*where* 是个字符串，指定了此自定义标记所被允许的 doc 注释类型。这些字符及其意义为 *a* (*all*: 所有地方都有效)、*p* (*package*: 包)、*t* (*type*: 类和接口)、*c* (*constructor*: 构造函数)、*m* (*method*) 以及 *f* (*field*: 字段)。

`-tag` 选项的第二个用途是指定标记被处理以及输出时的出现顺序。你可以在 `-tag` 选项后面加上标准标记的名称来指定这个顺序。自定义标记和 *taglet* 可以被包含在此标准 `-tag` 选项列表中。只有默认的 doclet 提供。

`-taglet classname`

指定“*taglet*”类的类名称来处理自定义标记。*taglet* 的编写在此不作说明。`-taglet` 标记可以用 `-tag` 标记来散布，以指定标记被处理和输出的顺序。只有默认的 doclet 提供。

`-tagletpath classpath`

指定以冒号分隔、由 JAR 文件或目录组成的列表，它们构成了用来搜索 *taglet* 类的 *classpath*。只有默认的 doclet 提供。

`-use`

针对列出类或包的用法的每个类和包，产生并插入链接到额外的文件中。

`-verbose`

在处理来源文件时显示额外的信息。

`-version`

从被产生的输出中的 `@version` 标记加上信息。此选项并不会告诉 *javadoc* 要输出它自己的版本编号。只有默认的 doclet 提供。

`-windowtitle text`

它定义了要输出在每个被产生文件的 `<Title>` 标记中的文本。此标题通常会作为 web 浏览器窗口的标题出现，并且会出现在历史记录与书签列表中。*text* 不应该包含 HTML 标记。请参阅 `-doctitle` 和 `-header`。只有默认的 doclet 提供。

## 环境

### CLASSPATH

此环境变量指定了 *javadoc* 用来寻找类文件和源文件的默认 *classpath*。它会被



-classpath和-sourcepath选项覆盖。关于classpath的进一步讨论,请参阅java和javac。

另可参阅 *java*、*javac*

## javah

C 语言原生 method 的 stub 产生器

### 提纲

```
javah [ options ] classnames
```

### 说明

*javah* 会产生 C 语言头文件和源文件 (.h 和 .c 文件), 当以 C 实现 Java 的原生 method 时, 就会用到它们。原生 method 接口在 Java 1.0 和 Java 1.1 之间有所改变。在 Java 1.1 及之前的版本中, *javah* 会产生针对老式原生 method 的文件; 在 Java 1.1 中, -jni 选项指出 *javah* 应该产生新式的文件。在 Java 1.2 及之后的版本中, 此选项是默认值。

本节只会说明使用 *javah* 的方式。关于如何使用 C 来实现 Java 的原生 method 的完整说明, 则超出了本书的范围。

### 选项

-bootclasspath

指定用来搜索系统类的路径。关于进一步的讨论请参阅 *javac*。Java 1.2 及之后的版本提供。

-classpath *path*

指定 *javah* 用来查询命令行上的类名的路径。此选项会覆盖所有由 CLASSPATH 环境变量指定的路径。在 Java 1.2 之前, 此选项可以指定系统类和扩展类的位置; 在 1.2 及之后的版本中, 它只能指定应用程序类的位置。请参阅 -bootclasspath。关于 classpath 的更进一步的讨论, 请参阅 *java*。

-d *directory*

指定 *javah* 用来存储所产生的文件的目录。默认情况下, *javah* 会把它们存储在当前目录中。此选项不可以与 -o 一起使用。

-force

使 *javah* 一定会写入输出文件, 即使那没有包含有用的内容。

-help

使 *javah* 显示简单的使用说明并退出。

**-Jjavaopt**

传递选项 *javaopt* 至 Java 解释器。

**-jni**

指出 *javah* 应该输出头文件以配合 Java Native Interface (JNI) 使用, 而不是配合旧的 JDK 1.0 原生接口。此选项在 Java 1.2 及之后的版本中是默认值。另请参阅 *-old*。Java 1.1 及之后的版本提供。

**-o outputfile**

将所有输出合并至单一文件, 而不是对每个指定的类创建独立的文件。

**-old**

输出 Java 1.0 风格的原生 *method* 的文件。在 Java 1.2 之前, 这是默认值。另请参阅 *-jni*。Java 1.2 及之后的版本提供。

**-stubs**

针对类文件而不是针对头文件来产生 *.c* 的 *stub* 文件。此选项只针对 Java 1.0 原生 *method* 接口。请参阅 *-old*。

**-trace**

指定 *javah* 应该把追踪输出命令包含于它所产生的 *stub* 文件中。在 Java 1.2 及之后的版本中, 此选项已过时且已被删除。你可以用 Java 解释器中的 *-verbose:jni* 选项来代替。

**-v, -verbose**

详细 (*verbose*) 模式。使 *javah* 输出关于进行中事项的消息。在 Java 1.2 及之后的版本中, *-verbose* 是同义词。

**-version**

使 *javah* 显示它的版本号。

## 环境

**CLASSPATH**

指定 *javah* 用来搜索指定类的默认 *classpath*。关于 *classpath* 的更进一步的讨论, 请参阅 *java*。

**另可参阅** *java*、*javac*

## javap

Java 类反汇编器

### 提纲

```
javap [ options ] classnames
```

### 说明

*javap* 会读取由命令行上的类名所指定的类文件，并列出由那些类所定义的人能看懂的 API。*javap* 也可以反汇编指定的类，显示它们所包含的 method 的 Java VM 字节码。

### 选项

-b

启用与 Java 1.1 版的 *javap* 输出的向后兼容性。此选项的存在是针对依赖于 *javap* 精确输出格式的程序。Java 1.2 及之后的版本提供。

-bootclasspath *path*

指定针对系统类的搜索路径。关于此极少使用的选项的信息请参阅 *javac*。Java 1.2 及之后的版本提供。

-c

显示每个被指定类的每个 method 的程序代码（即 Java VM 字节码）。此选项一定会反汇编所有的 method，而不管它们的可见度。

-classpath *path*

指定路径以供 *javap* 查询命令行上所命名的类。此选项会覆盖由 CLASSPATH 环境变量所指定的路径。在 Java 1.2 之前，这个自变量会指定针对所有系统类、扩展类以及应用程序类的路径。在 Java 1.2 及之后的版本中，它只指定应用程序的 classpath。也请参阅 -bootclasspath 和 -extdirs。关于 classpath 的更多信息，请参阅 *java* 和 *javac*。

-extdirs *dirs*

指定一个或多个针对扩展类进行搜索的目录。关于此极少使用的选项的信息，请参阅 *javac*。

-J*javaopt*

将选项 *javaopt* 传递给 Java 解释器。

-l

如果能在类文件中取得，就显示行号和局部变量列表。此选项通常只有在配合 -c 使用时才有用。默认情况下，*javac* 编译器不会在它的类文件中加上局部变量信息。请参阅针对 *javac* 的 -g 及相关选项。



-help

输出使用说明并退出。

-Jjavaoption

将指定的 *javaoption* 直接传递给 Java 解释器。

-package

显示在包层次可见的 (package-visible)、protected 以及 public 类成员，但不显示 private 成员。这是默认值。

-private

显示所有的类成员，其中包括 private 成员。

-protected

只显示 protected 和 public 成员。

-public

只显示指定类的 public 成员。

-s

使用内部的 VM 类型和 method 签名格式来输出类成员声明，而不使用更具可读性的源代码格式。

-verbose

详细模式。输出关于各个指定类的每个成员的额外信息（使用 Java 注释的形式）。

## 环境

CLASSPATH

指定针对应用程序类的默认搜索路径。-classpath 选项会覆盖这个环境变量。关于 classpath 的内容请参阅 *java*。

另可参阅 *java*、*javac*

## javaws

Java Web Start 启动器

### 提纲

```
javaws
javaws [ options ] url
```

### 说明

*javaws* 是针对 Java Web Start 网络应用程序启动器 (launcher) 的命令行接口。当在没

有添加url的情况下启用时, *javaws*会显示图形化的高速缓存监视程序,它能让高速缓存应用程序被启动以及让 Java Web Start 被配置。

如果 JNLP (Java Network Launching Protocol) 的 URL 被指定在命令行上, *javaws* 就会启动指定的应用程序。

## 选项

-association

允许在使用 -silent -import 的期间创建文件关联。

-codebase url

以指定的 url 覆盖 JNLP 文件中的 codebase。

-import

将指定的应用程序导入用户高速缓存中但不加以运行。

-offline

以离线模式运行。

-online

以联机模式启动。这是默认的行为。

-shortcut

允许在 -silent -import 的期间创建桌面快捷方式。

-silent

在与 -import 一起使用时, 此选项能避免 GUI 窗口出现。

-system

使用系统高速缓存。

-uninstall

从用户的高速缓存删除由 url 所标识的应用程序并退出。

-updateVersions

更新 *javaws* 配置文件 (例如在升级到新版的 Java 之后)。

-userConfig name [value]

设定布局特性 name; 或者, 如果 value 被指定, 就将 name 设定为指定的值。

-viewer

启动高速缓存监视应用程序。如果在不加上自变量的情况下调用 *javaws*, 这就是默认的行为。

-wait

直到被启动的应用程序退出时才退出。

-Xclearcache

清除用户高速缓存并退出。

-Xnosplash

不显示 Java Web Start 快闪画面。

## jconsole

## 图形化 Java 进程监控程序

### 提纲

```
jconsole [ options ]  
jconsole [ options ] pid  
jconsole [ options ] host:port
```

### 说明

*jconsole* 是针对内存、线程、类加载及其他由 `java.lang.management` 所提供的监控工具的图形化界面。它可以监控一个或多个本地或远程 Java 进程。只有以特殊的系统属性集启动的进程才能被监控。如果要允许 Java VM 能被本地监控，就应这样启动：

```
% jconsole -Dcom.sun.management.jmxremote=true
```

如果要允许 Java VM 能被远程监控，就应这样启动：

```
% jconsole -Dcom.sun.management.jmxremote.port=port
```

其中，*port* 是 *jconsole* 所要连接的远程端口。

你可以在不指定本地或远程进程的情况下启动 *jconsole*，并使用它的 Connection 菜单来建立连接。这是将 *jconsole* 连接至多个 Java 进程的唯一方法。

如果要在 *jconsole* 启动时连接至本地进程，只要把进程 id 列在命令行上就可以了。请查看 *jps* 来判定进程 id。

如果要在 *jconsole* 启动时连接至远程进程，就要在命令行上指定主机名称和通信端口号。此通信端口号应该与由目标进程的 `com.sun.management.jmxremote.port` 系统属性所指定的相同。



## 选项

-help

显示使用说明。

-interval=*n*

将更新期设定为 *n* 秒。默认值为 4。

-version

显示 *jconsole* 版本并退出。

另可参阅 *jps*、*jstat*

## jdb

Java 调试器

### 概要

```
jdb [ options ] class [ program options ]  
jdb connect options
```

### 说明

*jdb* 是 Java 类的调试器。它是基于文本、面向命令行的，而且具有像 C 和 C++ 程序所使用的 Unix *dbx* 或 *gdb* 调试器的命令行语法。

*jdb* 是用 Java 编写的，所以它会在 Java 解释器中运行。当 *jdb* 与 Java 类名一起被调用时，它会启动另一个 *java* 解释器副本，此副本会使用命令行上指定的所有解释器选项。新的解释器会以能让它与 *jdb* 通信的特殊选项来启动。新的解释器会加载指定的类文件，然后在执行第一个字节码之前停止并等待调试命令。

*jdb* 也可以对已在另一个 Java 解释器中运行的程序进行调试。要这么做需要把特殊的选项传递给 *java* 解释器和 *jdb*。请参阅以下的 -attach 选项。

### jdb 表达式语法

例如 *print*、*dump* 以及 *suspend* 这样的 *jdb* 调试命令，能让你以名称来引用类、对象、method、字段以及要被调试的程序中的线程。你可以用名称来引用类，是否加上它们的包名都可以。你也可以用名称来引用静态类成员。你可以用对象 ID 来引用个别的对象，而对象 ID 是八位的十六进制整数。或者，当你正在调试的类包含局部变量信息时，你通常可以使用局部变量名称来引用对象。你可以使用一般的 Java 语法来引用对象的字段和数组的元素；你也可以使用这个语法来编写相当复杂的表达式。在 Java 1.3 中，*jdb* 甚至支持使用标准 Java 语法的 *method* 调用。

## 选项

当加上指定的类文件来调用 *jdb* 时, 可以指定所有的 *java* 解释器选项。关于这些选项的说明, 请参阅 *java* 的参考页面。此外, *jdb* 支持以下选项:

`-attach [host:]port`

指出 *jdb* 应该连接到已在指定主机 (如果没有指定, 就是本地主机) 上运行的 Java VM, 并正在监听指定端口上的调试连接。Java 1.3 及之后的版本提供。

为了要以这种方式使用 *jdb* 来连接至运行中的 VM, VM 必须已用特殊的命令行选项启动。在 Java 1.3 和 1.4 中, 使用这些选项:

```
% java -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n
```

在 Java 5.0 中, 使用这些选项来代替:

```
% java -agentlib:jdwp=transport=dt_socket,address=8000,server=y,suspend=n
```

Java 调试结构允许复杂的解释器到调试器的连接选项, 而 *java* 和 *jdb* 提供了选项和子选项使其成为可能。对于这些选项的详细说明超出了本书的范围。

`-connect connector:args`

此选项对于将 *jdb* 连接至要被调试的进程, 提供了最普通和具有灵活性的方法。指定 *connector* (一个 Java 类) 的名称, 后面接着具有 `name=value` 形式、以逗号分隔的列表。Java 1.4 及之后的版本提供。关于可用的 *connector* 及它们的自变量, 请参阅 `-listconnectors`。

`-help`

显示的使用信息列出了所支持的选项。

`-launch`

当 *jdb* 启动时就启动指定的应用程序。这避免了明确地以 `run` 命令来加以启动的必要。Java 1.3 及之后的版本提供。

`-listconnectors`

列出可用的连接方法。每个 *connector* 都是一个 Java 类和自变量列表。Java 5.0 及之后的版本提供。请参阅 `-connect` 选项。

`-listen port`

监听 Java VM 的特定端口以连接至调试器。如果要让它运作, VM 就必须加上像这样的选项:

```
% java -agentlib:jdwp=transport=dt_socket,address=8000,server=n,suspend=y
```

Java 1.4 及之后的版本提供。

`-listenany`

和 `-listen` 选项一样, 但 `jdb` 会挑选一个端口来监听, 并列出端口号以在启动 Java 进程来调试时使用。Java 1.4 及之后的版本提供。

`-sourcepath path`

指定当尝试找出对应至要被调试的类文件的源文件时所搜索的位置。如果没有指定, `jdb` 就会根据默认使用 `classpath`。Java 1.3 及之后的版本提供。

`-tclient`

告诉 `jdb` 调用客户端的 Java 解释器版本。

`-tserver`

告诉 `jdb` 调用服务器端的 Java 解释器版本。

`-version`

显示 `jdb` 版本号并退出。

## 命令

`jdb` 能理解以下的调试命令。关于更多的信息请使用 `help` 命令。

`?或 help`

列出所有已支持的命令, 并对每条命令加上简短的说明。

`!!`

这条简短的命令会取代上一条输入的命令。后面可以加上额外的文字来跟在那条命令后面。

`catch [ exception-class ]`

每当有指定异常被抛出时就产生一个断点。如果没有指定异常, 此命令就会列出当前捕获的异常。使用 `ignore` 可以避免这些断点出现。

`classes`

列出所有已被载入的类。

`clear`

列出所有当前已设定的断点。

`clear class.method [(param-type...)]`

清除设定于指定类的指定 `method` 里的断点。

`clear [ class:line ]`

移除设定于指定类的指特定行的断点。

新华书店

开发工具



`cont`

继续执行。当当前线程停在断点时，就应该使用此命令。

`down [ n ]`

在当前线程的调用堆栈中向下移  $n$  帧 (frame)。如果没有指定  $n$ ，就向下移一帧。

`dump id...`

列出指定对象的所有字段的值。如果你指定了类名称，`dump`就会显示出所有类(静态) `method` 和变量，并显示超类和已实现接口的列表。对象和类可以通过名称或它们的八位十六进制 ID 号来指定。线程也可以用简便的 `t@thread-number` 来指定。

`exit` 或 `quit`

结束 `jdb`。

`gc`

运行内存回收以强制回收未被使用的对象。

`ignore exception-class`

不要把特定的异常当成断点。此命令会关闭 `catch` 命令。此命令不会使 Java 解释器忽略异常，它只是告诉 `jdb` 要忽略异常。

`list [ line-number ]`

列出指定行及其前后几行的源代码。如果没有指定行号，就使用当前线程的当前堆栈帧的行号。被列出的行是来自当前线程的当前堆栈帧的源文件。使用 `use` 命令可以告诉 `jdb` 要到哪里找源文件。

`list method`

显示指定 `method` 的源代码。

`load classname`

将指定类载入 `jdb`。

`locals`

显示针对当前堆栈帧的局部变量列表。Java 程序代码必须使用 `-g` 选项来编译，以便于包含局部变量信息。

`methods class`

列出指定类的所有 `method`。使用 `dump` 列出对象的实例变量或类的类(静态)变量。

`print id...`

列出指定项的值。各个入口项是类、对象、字段或局部变量，而且可以用名称或八位十六进制 ID 号来指定。你也可以用特殊的 `t@thread-number` 语法来引用线程。`print` 命令会通过调用它的 `toString()` `method` 来显示对象的值。

`next`

执行源代码的当前行，其中包括它做的所有 `method` 调用。另可参阅 `step`。

`resume [ thread-id... ]`

继续执行指定的线程。如果没有指定线程，所有挂起的线程就会恢复。另请参阅 `suspend`。

`run [ class ] [ args ]`

运行指定类的 `main()` `method`，把指定的自变量传递给它。如果没有指定类或自变量，就会使用 `jdb` 命令行上指定的类和自变量。

`step`

运行当前线程当前所在的行并再次停止。如果这行调用了 `method`，就会进入那个 `method` 然后停止。另请参阅 `next`。

`stepi`

执行一条单一 Java VM 指令。

`step up`

运行直到当前 `method` 返回至其调用者，并再次停止。

`stop`

列出当前的断点。

`stop at class:line`

在指定类的指定行设置断点。程序会在运行到这行时停止。可以使用 `clear` 来删除断点。

`stop in class.method [(param-type...)]`

在指定类的指定 `method` 的开始设置断点。程序会在进入此 `method` 时停止。可以使用 `clear` 来删除断点。

`suspend [ thread-id... ]`

挂起指定的线程。如果没有指定线程，就挂起所有运行中的线程。使用 `resume` 可以重新启动它们。

`thread thread-id`

将当前线程设定为指定的线程。此线程会由一些其他的 `jdb` 命令隐含地使用。

`threadgroup name`

设定当前线程组。

`threadgroups`

列出所有在正在调试的 Java 解释器会话上运行的线程组。

`threads [ threadgroup ]`

列出具名线程组中的所有线程。如果没有指定线程组，就列出当前线程组（由 `threadgroup` 指定）中的所有线程。

`up [ n ]`

在当前线程的调用堆栈中向上移  $n$  帧。如果没有指定  $n$ ，就向上移一帧。

`use [ source-file-path ]`

设定 `jdb` 所使用的路径，以查询要被调试的类的源文件。如果没有指定路径，就显示当前源路径。

`where [thread-id] [all]`

显示指定线程的堆栈踪迹。如果没有指定线程，就显示当前线程的堆栈踪迹。如果指定了 `all`，就显示所有线程的堆栈踪迹。

`wherei [thread-idx]`

显示指定或当前线程的堆栈踪迹，其中包括详细的程序计数信息。

## 环境

### CLASSPATH

指定了目录、ZIP 文件以及 JAR 归档文件的有序列表（在 Unix 系统中用冒号分隔，在 Windows 系统中用分号分隔），`jdb` 会以此列表来寻找类定义。当使用这个环境变量指定路径时，`jdb` 一定会隐式地将系统类的位置附加到此路径的结尾。如果没有指定这个环境变量，默认的路径就是当前目录和系统类。此变量会被 `-classpath` 选项覆盖。

另可参阅 `java`

## jinfo

显示 Java 进程的配置

### 提纲

```
jinfo [ options ] pid           // 在本地进程上的信息
jinfo [ options ] executable core // 来自 core 文件的信息
jinfo [ options ] {process-name}@hostname // 来自远程进程的信息
```

### 说明

`jinfo` 会列出运行中的 Java 进程或核心文件的系统特性和 JVM 命令行选项。`jinfo` 的启动方式有以下三种：



- 指定运行于本地的 Java 进程的进程 id 来取得关于它的配置信息。请参阅 *jps* 来列出本地的进程。
- 如果要从核心文件取得事后检查 (post-mortem) 配置信息, 就在命令行上指定产生核心文件的 Java 可执行文件和核心文件本身。
- 如果要取得关于远程运行的 Java 进程的配置信息, 就指定远程主机的名称并选择性地前面加上远程进程名称。在远程主机上必须运行 *jsadepugd*。

在 Java 5.0 中, *jinfo* 是实验性的、未被支持的, 而且不是在所有平台都可用的。

## 选项

这些选项是互斥的, 只能指定其中一个。

*-flags*

只列出 JVM 标记, 而不列出系统特性。

*-help*、*-h*

列出辅助说明信息。

*-sysprops*

只列出系统特性, 而不列出 JVM 标记。

另可参阅 *jps*、*jsadepugd*

---

## jmap

显示内存使用状况

### 提纲

```
jmap [ options ] pid           // 本地进程
jmap [ options ] executable core // 核心文件
jmap [ options ] [process-name@]hostname // 远程进程
```

### 说明

*jmap* 会列出本地或远程 Java 进程或 Java 核心文件的内存使用情况的信息。依据配合调用的选项, *jmap* 可以显示的内存使用状况报表有四种。相关细节请参阅选项章节。*jmap* 可以用三种方式来启动:

- 指定运行于本地的 Java 进程的进程 id 来取得关于它的配置信息。请参阅 *jps* 来列出本地进程。

- 如果要从核心文件取得事后检查 (post-mortem) 配置信息, 就在命令行上指定产生核心文件的 Java 可执行文件和核心文件本身。
- 如果要取得关于远程运行的 Java 进程的配置信息, 就指定远程主机的名称并选择性地前面加上远程进程名称和 @ 符号。在远程主机上必须运行 *jsadebugd*。

在 Java 5.0 中, *jmap* 是实验性的、未被支持的, 而且不是在所有平台都可用的。

## 选项

在不加上选项进行调用时, *jmap* 会列出 VM 所加载的共享对象或函数库的内存映射 (map)。其他的报告可以使用以下选项来产生。这些选项是互斥的, 只能指定其中一个。

-heap

显示堆内存使用情况的汇总。

-help, -h

列出辅助说明信息。

-histo

以类显示堆使用情况的柱状图。

-permstat

显示已加载的类所使用的内存, 以类加载程序来分组。

另可参阅 *jps*、*jsadebugd*

---

## jps

列出 Java 进程

## 提纲

```
jps [ options ] [ hostname[:port ] ]
```

## 说明

*jps* 会列在本地主机或在指定远程主机上运行的 Java 进程。如果指定了远程主机, 那台主机就必须运行 *jstatd* 后台进程 (daemon)。针对每个 Java 进程, 它会显示进程正在执行的类或 JAR 文件的进程 id 和名称。进程 id 会被其他的一些 Java 工具使用, 例如 *jconsole*、*jstat* 以及 *jmap*。

## 选项

以下的选项会更改默认的 *jps* 显示项目。除了 -q 之外, 单一字母的选项可以结合成单一命令行自变量, 例如 -lmv:

-help

显示使用状况信息。

-l

列出在每个 Java 进程中运行的主要类的完整包名称或 JAR 文件的完整路径。

-m

列出传递给每个 Java 进程的 main() method 的自变量。

-q

只列出 Java 进程标识符，而不列出应用程序名称或任何额外的信息。

-v

列出通过每个 Java 进程传递给 Java 解释器的自变量。

-V

列出通过标记文件（例如 *.hotspotrc*）传递给解释器的自变量。

另可参阅 *jstatd*

---

## jsadbugd

用于远程调试的 daemon 进程

### 提纲

```
jsadbugd pid [ process-name ]           // 运行中的进程
jsadbugd executable core [ process-name ] // 核心文件
```

### 说明

*jsadbugd* 是个服务器进程，它允许在本地 Java 进程或核心文件上的 *jinfo*、*jmap* 和 *jstack* 的远程调用。通过在命令行上指定运行中的 Java 进程或可执行文件的进程 id 和核心文件对，就可以调用 *jsadbugd*。如果同一台主机上要同时运行多个 *jsadbugd* 服务器，就在这些自变量后面加上可识别的进程名称，让远程的客户端用来识别所需的进程。

*jsadbugd* 会启动 *rmiregistry* 服务器。

在 Java 5.0 中，*jsadbugd* 是实验性的、未被支持的，而且不是在所有平台都可用的。

另可参阅 *jinfo*、*jmap*、*jstack*



**jstack**

显示 Java 进程的堆栈踪迹

**提纲**

```
jstack [ options ] pid           // 本地进程
jstack [ options ] executable core // 核心文件
jstack [ options ] [process-name@]hostname // 远程进程
```

**说明**

*jstack* 会列出运行于指定 Java 进程中的各个 Java 线程的堆栈踪迹。*jstack* 的启动方式有三种：

- 指定运行于本地的 Java 进程的进程 id 来取得关于它的配置信息。请参阅 *jps* 来列出本地进程。
- 如果要从核心文件取得事后检查配置信息，就在命令行上指定产生核心文件的 Java 可执行文件和核心文件本身。
- 如果要取得有关远程运行的 Java 进程的配置信息，就指定远程主机的名称并选择性地前面加上远程进程名称和 @ 符号。在远程主机上必须运行 *jsadefugd*。

在 Java 5.0 中，*jstack* 是实验性的、未被支持的，而且不是在所有平台都可用。

**选项**

-help、-h

列出辅助说明信息。

-m

以“混合模式”显示堆栈踪迹，也就是同时显示 Java 和原生 method 堆栈帧。如果没有加上这个选项，就是默认为只显示 Java 堆栈帧。

**另可参阅**     *jps*、*jsadefugd*

**jstat**

Java VM 统计数据

**提纲**

```
jstat [options] pid [ interval[s|ms] [ count ] ]
jstat [options] pid@hostname[:port] [ interval[s|ms] [ count ] ]
```

## 说明

*jstat* 会探测运行中的 JVM 一次或重复多次，并显示关于它的类加载、实时编译、内存回收性能的统计数据。所显示的信息类型是由 *option* 来指定。要被探测的本地进程会通过它的进程 id 来指定，例如由 *jps* 返回的 id。远程 Java 进程可以通过指定远程进程 id、远程主机名称以及远程主机的 *rmiregistry* 服务器上所运行的通信端口编号（如果不是默认的 1099）来探测。远程主机也必须运行 *jstatd* 服务器。

默认情况下，*jstat* 会探测指定的 Java VM 一次。你也可以指定以毫秒或秒为单位的探测间隔来让它重复探测。如果你这么做，就可以额外指定要实施的探测总次数。

*jconsole* 可以报告许多与 *jstat* 相同的统计数据，但是使用图形形式来加以显示而不是表格形式。在 Java 5.0 中，*jinfo* 是实验性的、未被支持的，而且不是在所有平台都可用。

## 选项

-help

显示辅助说明信息。

-options

显示 *jstat* 所能显示的报告类型的列表。每次运行 *jstat*，你都必须使用其中一个被列出的选项。

-version

显示 *jstat* 版本信息并退出。

-h n

当 *jstat* 重复探测 Java 进程时，此选项指定了多久就要在输出中重复表格的表头。此选项必须接在以下一种报告类型选项后面。

-t

对由 *jstat* 产生的报告增加 Timestamp 栏。此栏显示从目标 Java 进程启动后所经过的时间（以秒为单位）。

以下选项指定由 *jstat* 报告的统计数据类型。除非你加上 -help、-options 或 -version 来运行 *jstat*，否则你应该精确地指定这些选项的其中一个，而且它必须是命令行上的第一个选项。大部分的选项会产生内存回收细节的详细报告。关于这些报告的解释，请参阅 Sun 的工具说明文件（这是 JDK 文件包的一部分）。

-class

报告已加载的类的数量和它们的大小（以 kilobyte 为单位）。

-compiler

报告已执行的实时编译的数量以及所花的时间。

-gc

报告堆内存回收统计数据。

-gccapacity

报告内存回收程序的各个内存区的容量信息。

-gccause

就像 -gcutil 所报告的，但加上关于最近内存回收的成因的信息。

-gcnew

报告关于内存回收程序的“新一代 (new generation)”内存池的信息。

-gcnewcapacity

报告内存回收程序的“新一代”内存池的容量信息。

-gcold

报告内存回收程序的“旧时代”和“永久代” (permanent) 内存池的信息。

-gcoldcapacity

报告内存回收程序的“旧时代”内存池的信息。

-gcpermcapacity

报告内存回收程序的“永久代”的信息。

-gcutil

报告内存回收的汇总数据。

-printcompilation

报告关于实时编译的额外信息，其中包括已编译的 method 的名称。

**另可参阅**     *jconsole*、*jps*、*jstatd*

---

**jstatd**

jstat daemon

**提纲**

*jstatd options*

**说明**

*jstatd* 是服务器，它提供关于本地 Java 进程的信息给运行于远程主机上的 *jps* 和 *jstat* 程序。



*jstatd*使用了RMI而且需要特殊的安全权限才能顺利运行。如果要启动*jstatd*，可以创建以下文件并把它命名为*jstatd.policy*：

```
grant codebase "file:${java.home}../lib/tools.jar {  
    permission java.security.AllPermission  
}
```

此策略会将所有的权限授予任何由JDK的名为*tools.jar*的JAR文件所载入的类。如果用这个策略来启动*jstatd*，可以使用这行命令：

```
% jstatd -J-Djava.security.policy=jstatd.policy
```

如果已有*rmiregistry*服务器在运行，*jstatd*就会使用它；否则，它会创建它自己的RMI注册（registry）。

## 选项

-n *rminame*

将*jstatd*远程对象与RMI registry中的*rminame*名称绑定。默认的名称是“JstatRemoteHost”，这就是*jps*和*jstat*所寻找的。如果要使用这个选项，远程的*jps*和*jstat*调用中就必须使用*rminame*。

-nr

告诉*jstatd*如果没有已运行的内部RMI registry，不要启动内部RMI registry。

-p *port*

寻找*port*上已有的RMI registry，如果没有找到已有的registry，就在那个通信端口上启动一个。

另可参阅 *jps*、*jstat*

## keytool

密钥与证书管理工具

### 提纲

*keytool* command options

### 说明

*keytool*管理并操作*keystore*，*keystore*是公钥、密钥和公钥证书的存储库。*keytool*定义了各种命令，用于产生密钥、将数据导入*keystore*以及导出并显示*keystore*数据。密钥和证书是使用不区分大小写的名称或别名来存储在*keystore*中。*keytool*使用这个别名来引用密钥或证书。

*keytool*的第一个选项总是指定所要执行的基本命令。后续的选项提供了关于如何执行命令的细节。只有命令才一定要被指定。如果命令所需的选项不具有默认值, *keytool*就会以交互的方式来提示你输入值。

## 命令

### -certreq

针对指定的别名产生PKCS#10格式的证书签署请求(certificate signing request)。此请求会被写入指定的文件或标准输出流。此请求应该被送到认证机构(certificate authority, CA), 它会验证请求者并回送加上签名的证书来证明请求者的公钥是真的。对于这个已加上签名的证书接着可以用 *-import* 命令来导入 *keystore*。此命令使用了以下选项: *-alias*、*-file*、*-keypass*、*-keystore*、*-sigalg*、*-storepass*、*-storetype* 和 *-v*。

### -delete

从指定 *keystore* 删除指定的别名。此命令使用了以下选项: *-alias*、*-keystore*、*-storepass*、*-storetype* 以及 *-v*。

### -export

将证书及相关的指定别名写入指定文件或标准输出。此命令使用了以下选项: *-alias*、*-file*、*-keystore*、*-rfc*、*-storepass*、*-storetype* 和 *-v*。

### -genkey

产生一对公共/私有密钥以及针对公钥的自签(self-signed)的X.509证书。自签证书本身通常没什么用, 所以这个命令后面通常会接着 *-certreq*。此命令使用了以下选项: *-alias*、*-dname*、*-keyalg*、*-keypass*、*-keysize*、*-keystore*、*-sigalg*、*-storepass*、*-storetype*、*-v* 和 *-validity*。

### -help

列出所有可用的*keytool*命令及它们的选项。此命令不会与任何其他命令一起使用。

### -identitydb

从利用已废除的*javakey*程序所管理的原有身份数据库中读取密钥和证书并把它们存储到 *keystore* 中, 以便让它们能被 *keytool* 操作。此身份数据库是从指定的文件中被读取的, 如果没有指定文件, 就会从标准输入来读取。密钥和证书会被写入指定的 *keystore* 文件中, 如果它还不存在, 就会自动创建。此命令使用了以下选项: *-file*、*-keystore*、*-storepass*、*-storetype* 和 *-v*。

### -import

从指定的文件或从标准输入中读取证书或PKCS#7格式的证书链(certificate

chain), 并把它当成可信的证书来以指定的别名存储在 *keystore* 中。此命令使用了以下选项: `-alias`、`-file`、`-keypass`、`-keystore`、`-noprompt`、`-storepass`、`-storetype`、`-trustcacerts` 和 `-v`。

#### `-keyclone`

复制指定别名的密钥存储入口, 并把它存储在 *keystore* 中的一个新别名中。此命令使用了以下选项: `-alias`、`-dest`、`-keypass`、`-keystore`、`-new`、`-storepass`、`-storetype` 和 `-v`。

#### `-keypasswd`

变更用来加密与指定别名相关联的私钥的密码。此命令使用了以下选项: `-alias`、`-keypass`、`-new`、`-storetype` 和 `-v`。

#### `-list`

(在标准输出上) 显示与指定别名相关联的证书的特征。如果加上 `-v` 选项, 就以人能看懂的格式列出证书的细节; 如果加上 `-rfc`, 就以机器可理解的、可打印的编码格式列出证书的内容。此命令使用了以下选项: `-alias`、`-keystore`、`-rfc`、`-storepass`、`-storetype` 和 `-v`。

#### `-printcert`

显示从指定文件或标准输入中读取的证书的内容。与大部分的 *keytool* 命令不同, 这个命令没有使用密钥存储。此命令使用了以下选项: `-file` 和 `-v`。

#### `-selfcert`

针对与指定别名称相关联的公钥创建自签证书, 并用它取代已与那个别名相关联的任何证书或证书链。此命令使用了以下选项: `-alias`、`-dname`、`-keypass`、`-keystore`、`-sigalg`、`-storepass`、`-storetype`、`-v` 和 `-validity`。

#### `-storepasswd`

变更保护密钥存储整体完整性的密码。新的密码在长度上至少要有六个字符。此命令使用了以下选项: `-keystore`、`-new`、`-storepass`、`-storetype` 和 `-v`。

## 选项

以下列出的各种选项都可以传递给各种 *keytool* 命令。这些选项中有很多有合理的默认值。对于任何不具有默认值且未被指定的选项, *keytool* 会以交互的方式提示我们输入值。

#### `-alias name`

指定密钥存储中用到的别名。默认值为 “mykey”。



`-dest newalias`

为 `-keyclone` 命令指定新的别名（目的别名）。如果没有指定，`keytool` 就会提示我们输入值。

`-dname X.500-distinguished-name`

指定 X.500 专有名称（distinguished name）要出现在由 `-selfcert` 或 `-genkey` 所产生的证书上。专有名称是个高度限定的名称，在全球都是唯一的。例如：

CN=David Flanagan, OU=Editorial, O=OReilly, L=Cambridge, S=Massachusetts, C=US

如果没有指定识别名称，`keytool` 的 `-genkey` 命令就会提示输入。如果没有指定替代名称，`-selfcert` 命令就会使用当前证书的专有名称。

`-file file`

为 `keytool` 命令指定输入或输出文件。如果没有指定，`keytool` 就会从标准输入中读取或写入到标准输出。

`-keyalg algorithm-name`

配合 `-genkey` 使用，以指定要产生哪种类型的加密密钥。在 Sun 所默认的 Java 实现中，唯一支持的算法是“DSA”，如果此选项被忽略，这就是默认值。

`-keypass password`

指定在加密密钥存储中的私钥所用的密码。如果没有指定这个选项，`keytool` 会先尝试 `-storepass password`。如果不成功，它就会提示我们要输入适当的密码。

`-keysize size`

配合 `-genkey` 命令一起使用，以指定生成的密钥以位为单位的长度。如果没有指定，默认值为 1024。

`-keystore filename`

指定密钥存储文件的位置。如果没有指定，就会使用位于用户根目录中名为 `.keystore` 的文件。

`-new new-password-or-alias`

配合 `-keyclone` 命令一起使用以指定新的别名，以及配合 `-keypasswd` 和 `-storepasswd` 一起使用以指定新的密码。如果没有指定，`keytool` 就会提示我们要输入这个选项的值。

`-noprompt`

配合 `-import` 命令一起使用，以在无法针对导入的证书创建信任链时禁止和用户之间的交互。如果没有指定这个选项，`-import` 命令就会提示用户。

`-rfc`

配合 `-list` 和 `-export` 命令一起使用, 以指定证书的输出应该要使用由 RFC 1421 所指定的可打印编码格式。如果没有指定这个选项, `-export` 就会以二进制格式输出证书, 而 `-list` 会只列出证书的特征。此选项在 `-list` 命令中不可以与 `-v` 结合。

`-sigalg algorithm-name`

指定签署证书的 digital signature 算法。如果省略, 则此选项的默认值会取决于底层公钥的类型。如果它是 DSA 密钥, 则默认的算法就是 “SHA1withDSA”; 如果它是 RSA 密钥, 则默认的签名算法就是 “MD5withRSA”。

`-storepass password`

指定用来保护整个密钥存储文件的完整性的密码。这个密码也作为本身没有指定 `-keypass` 的所有私钥的默认密码。如果没有指定 `-storepass`, `keytool` 就会加以提示。此密码的长度至少为六个字符。

`-storetype type`

指定所要使用的密钥存储类型。如果没有指定这个选项, 就会从系统安全属性文件中取得默认值。通常, 默认值为 “JKS” —— Sun 的 Java Keystore 类型。

`-trustcacerts`

配合 `-import` 命令一起使用, 以指示包含于 `jar/lib/security/cacerts` 文件的密钥存储中的自签证书应该被当成可信任的。如果忽略这个选项, `keytool` 就会忽略那个文件。

`-v`

详细模式, 只要定义了它, 就会让许多的 `keytool` 命令产生额外的输出。

`-validity time`

配合 `-gencert` 和 `-selfcert` 命令一起使用, 以指定所产生的证书的有效期 (单位为天)。如果没有指定, 默认值为 90 天。

另可参阅 `jarsigner`、`policytool`

---

## native2ascii

将文字转换为具有 Unicode 转义字符的 ASCII

### 提纲

```
native2ascii [ options ] [ inputfile [ outputfile ] ]
```

### 说明

`native2ascii` 是个简单的程序, 它读取使用地区编码的文本文件 (通常是 Java 源文件),

并把它转换为Java语言规范所允许的Latin-1-plus-ASCII-encoded-Unicode形式。当你必须编辑Java程序代码文件,但又没有可以处理文件编码的编辑器时,这是很有帮助的。

*inputfile*和*outputfile*是选择性的。如果没有指定,就会使用标准输入和标准输出,这使得 *native2ascii* 很适合用在管道中。

## 选项

`-encoding encoding-name`

指定源文件所使用的编码方式。如果没有指定这个选项,就会从 *file.encoding* 系统属性中取得编码方式。

`-reverse`

指定转换应该要反向进行——从已编码的 `\uxxxx` 字符转换为原生编码方式的字符。

## 另可参阅

`java.io.InputStreamReader`, `java.io.OutputStreamWriter`

---

## pack200

压缩 JAR 文件

## 提纲

`pack200 [options] outputfile jarfile`

## 说明

*pack200*会使用由JSR 200和标准gzip压缩算法的定义的压缩算法来紧密压缩JAR文件。请注意,命令行上的输出文件是被指定在输入的JAR文件之前。

## 基本选项

所有的*pack200*选项都同时以详细形式和单字母形式存在,详细形式是以双连字符开始,而单字母形式是以单连字符开始。当选项必须要有值时,这个值在详细形式中应该用等号与选项分隔,而且不可以有空格;而在单字母形式中应该要紧接在选项后面,中间不可以有空格或标点符号。

`--config-file=file`, `-ffile`

从指定的配置文件读取选项。*file*应该是具有 `name=value` 格式的 `java.util.Properties` 文件。所支持的特性名称与列于此处的详细形式的选项名称相同,但连字符转换为句号。



--effort=value、-Evalue

指定尝试压缩 JAR 文件的程度。*value* 必须是介于 0 和 9 之间的数字。0 代表完全不压缩，只是产生输入的 JAR 文件的副本。默认值为 5。

--help、-h

显示辅助说明信息并退出。

--log-file=file、-lfile

将输出记录到文件。

--no-gzip、-g

告诉 *pack200* 不要对已压缩的 JAR 文件使用 gzip 压缩。如果你想要使用不同的压缩过滤程序，例如 *bzip2*，就使用这个选项。默认值为 --gzip。

--no-keep-file-order、-o

允许 *pack200* 重新排序 JAR 文件的元素。--keep-file-order 为默认值。

--quiet、-q

抑制输出信息。

--pass-file=file、-Pfile

跳过指定的文件不进行压缩。如果 *file* 是以 / 结束，目录中所有的文件都会被跳过而不被压缩。这个选项可以被指定多次。

--repack、-r

压缩指定的 JAR 文件，然后立刻解压缩。在这样的情况下，在命令行上所指定的 *outputfile* 就应该是 JAR 文件的名称。在使用 *jarsigner* 来签名之前，在 JAR 文件上进行压缩 / 解压缩循环是很重要的，因为压缩 / 解压缩循环会重排类文件的一些内部元素，并使得 JAR 文件清单中的所有数字签名或校验码变为无效。

--strip-debug、-G

从 Java 类文件永久移去调试属性，而不是对它们进行压缩。这会使得要对最后产生的 JAR 文件进行调试会较为困难。

--verbose、-v

显示更多的输出信息。

--version、-V

显示版本编号并退出。

## 高级压缩选项

以下选项提供了在由 *pack200* 所执行的压缩上的精细控制。

--deflate-hint=value、-Hvalue

指定`pack200`是否应该保留输入的JAR文件中各个项目的紧缩状态。默认的`value`为`keep`, 指将状态保留。如果`value`是`true`, 就会在已压缩的归档文件中放入一个提示, 指示解压缩程序在解压缩所有项目后应该再把它们紧缩。如果`value`是`false`, 就会在已压缩的归档文件中放个提示, 指示解压缩程序应该在不紧缩JAR文件中各个项目的情况下加以存储。`value`使用`true`或`false`可以稍微降低压缩文件的大小, 因为紧缩提示不必针对每个项目来存储。

--modification-time=value、-mvalue

在使用默认的`keep`作为`value`时, `pack200`会留传JAR文件中每个项目的修改次数。如果你指定了`latest`, 那就只会留传最近的修改时间, 而且会在所有的项目解压缩时被应用。

--segment-limit=n、-Sn

将目标片段(segment)大小设定为`n`。`pack200`文件可以被分割为独立的压缩片段, 以便于降低解压缩程序所需的内存数量。此选项会设定各个片段的近似大小。默认值为1兆字节。值-1会产生单一型片段, 而值0会对每个类文件产生一个片段。较大的片段大小会产生较佳的压缩率, 但会需要额外的内存来解压缩。

--unknown-attribute=action、-Uaction

指示`pack200`应该如何处理未知的类文件属性。默认的`action`是`pass`, 这指示整个类文件会在不压缩的情况下被留传。`action`若是`error`, 则是指示`pack200`应该产生错误信息。`action`若是`strip`, 则是说此属性应该要从类文件中移去。

--class-attribute=name=action、-Cname=action

--code-attribute=name=action、-Dname=action

--field-attribute=name=action、-Fname=action

--method-attribute=name=action、-Mname=action

这四个选项指定`pack200`应该如何处理特定的具名类、字段、method以及类文件中的程序代码属性。属性的名称是由`name`指定。`action`可以是由--unknown-attribute选项所支持的`pass`、`strip`和`error`值, 也可以是指示属性应该如何被压缩的“布局字符串”。关于在布局语言上的细节, 请参阅Pack200规范说明书。这些选项可以重复, 以指定对多个属性的处理。

另可参阅 `unpack200`

## policytool

## 策略文件的创建与管理工具

### 提纲

policytool

### 说明

*policytool* 会显示 Swing 用户界面，使得安全策略配置文件的编辑变得容易。Java 的安全结构是以策略文件为基础，它指定了各种源代码允许的权限集。默认情况下，Java 的安全策略是由存储于 *jre/lib/security/java.policy* 文件中的系统策略文件和存储于用户根目录的 *.java.policy* 策略文件中的用户策略文件所定义的。系统管理员和用户可以用文本编辑器来编辑这些文件，但文件的语法有些复杂，所以使用 *policytool* 来定义和编辑安全策略通常会比较简单。

### 选择要编辑的策略文件

当 *policytool* 启动时，默认情况下，它会开启用户根目录中的 *.java.policy* 文件。使用 File 菜单中的 New、Open 和 Save 命令来分别创建新的策略文件、开启已存在的文件以及存储编辑过的文件。

### 编辑策略文件

*policytool* 的主窗口会显示包含于策略文件中的项目列表。每个项目指定了一个程序代码来源和该来源的程序代码所拥有的权限。此窗口也包含了一些按钮，能让你增加新项目、编辑已存在的项目或删除策略文件中的项目。如果你新增或编辑一个项目，则 *policytool* 会开启新的窗口来显示那个策略项目的细节。

随着 JAAS API 在 Java 1.4 中加入核心 Java 平台，*policytool* 允许 Principal 的规范，那是组要被授予的权限。

每个策略文件都有一个相关联的密钥存储，在验证 Java 源代码的数字签名时，策略文件就会从密钥存储取得所需的证书。你通常可以依赖默认的密钥存储，但如果你必须明确地为策略文件指定密钥存储，那么可以使用 *policytool* 主窗口的 Edit 菜单中的 Change Keystore 命令。

### 新增或编辑策略项目

策略项目编辑器窗口显示了策略项目的代码来源，以及与该代码来源相结合的权限列表。它也包含了一些按钮，能让你增加新的权限、删除权限或编辑已存在的权限。



当定义新的策略项目时,第一步就是要指定程序代码来源。程序代码来源是由URL所定义的,就是下载程序代码和/或必须出现在程序代码中的数字签名列表的URL。通过输入URL和/或以逗号分隔的别名列表,就可以指定这些值。这些别名标识了在密钥存储中与策略文件相关联的可信任的证书。

在定义了策略项目的代码来源之后,你必须定义要被授权给来自那个来源的程序代码所拥有的权限。你可以使用Add Permission和Edit Permission按钮来增加和编辑权限。这些按钮会弹出另一个 *policytool* 窗口。

## 定义权限

如果要在权限编辑器窗口定义权限,首先要从Permission下拉菜单选择所要的权限类型,然后从Targer Name菜单选择适当的目标值,在此菜单中的选项是取决于你所选择的权限。有些类型的权限,例如FilePermission,没有固定的可能目标集合,那么你通常必须输入你要的目标。例如,你可以输入“/tmp/-”来指定/tmp目录,输入“/tmp/\*”来指定那个目录以及子目录中的所有文件。关于它们所支持的目标的说明,请参阅各个Permission类的说明文件。

依据你所选择的权限类型,你也可以从Actions菜单选择一个或多个动作值。当你已选择权限和适当的目标与动作值时,就可以单击Okay按钮来关闭窗口了。

另可参阅 *jarsigner*、*keytool*

---

## serialver

类版本编号产生器

### 提纲

```
serialver [ -classpath path ] classnames...  
serialver [ -classpath path ] -show
```

### 说明

*serialver*会显示一个类或一些类的版本编号。这个版本编号是用于序列化:每当类的序列化格式改变时,版本编号就必须改变。

如果被指定的类声明了long serialVersionUID常量,那么该字段的值就会被显示。否则,具有唯一性的版本编号就会通过将Secure Hash Algorithm (SHA)应用至由类定义的API上来计算出来。这个程序对于计算类的初始唯一版本编号很有用,接着那个编号会被声明为类中的常量。*serialver*的输出是一行合法的Java程序代码,适合粘贴到类定义中。

## 选项

`-classpath path`

指定类的搜索路径。

`-show`

当指定 `-show` 选项时, *serialver* 会显示简单的图形界面, 能让用户一次输入一个类名并取得它的序列化 UID。当使用 `-show` 时, 命令行上不可以指定类名。

## 环境

CLASSPATH

*serialver* 是用 Java 编写的, 所以它对 CLASSPATH 环境变量的敏感度与 *java* 解释器相同。被指定的类会以相对于此 classpath 的方式来查询。

另可参阅 `java.io.ObjectStreamClass`

## unpack200

解压缩 JAR 文件

### 提纲

```
unpack200 [options] packedfile jarfile
```

### 说明

*unpack200* 会解压缩被 *pack200* 工具压缩的 JAR 文件并选择性地以 *gzip* 来压缩。在命令行上指定压缩文件的名称和要对它解压缩的 JAR 文件的名称。

因为 *unpack200* 被作为 Java 安装程序的一部分, 所以它是原生应用程序, 可以在没有 Java 解释器的情况下在系统上运行。

### 选项

所有的 *unpack200* 选项都同时以详细形式和单字母形式存在, 详细形式以双连字符开始, 而单字母形式以单连字符开始。当选项必须要有值时, 这个值在详细形式中应该要用等号与选项分隔而且不可以有空格; 在单字母形式中应该要紧接在选项后面, 中间不可以有空格或标点符号。

`--deflate-hint=value -Hvalue`

指定 *unpack200* 是否应该压缩最后产生的 JAR 文件里的个别项目。value 必须是 `true`、`false` 或 `keep`。默认值为 `keep`, 这指示每个 JAR 项目应该有与原来的 JAR 文件中相同的压缩形式。

--help、-h

显示辅助说明信息并退出。

--log-file=file、-lfile

将输出记录至文件。

--quiet、-q

抑制输出信息。

--remove-pack-file、-r

在对压缩文件解压缩后，就将它删除。

--verbose、-v

显示更多的输出信息。

--version、-V

显示版本编号并退出。

另可参阅 *jar*、*pack200*

